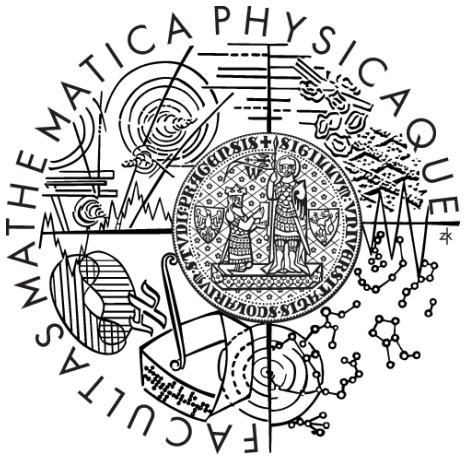
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Ľuboš Píš

Generování kódu ze stavového modelu UML

Katedra softwarového inženýrství

Vedoucí diplomové práce: doc. Ing. Karel Richta, CSc.

Studijní program: Informatika

Studijní obor: Softwarové systémy

Praha 2011

Ďakujem týmto vedúcemu diplomovej práce doc. Ing. Karlovi Richtovi, CSc. za jeho odborné rady a metodické vedenie pri spracovaní tejto diplomovej práce.

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle 60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Generování kódu ze stavového modelu UML
Autor: Ľuboš Píš
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: doc. Ing. Karel Richta, CSc.
e-mail vedoucího: richta@ksi.ms.mff.cuni.cz

Abstrakt: Táto práca sa zaoberá implementáciou vhodného algoritmu generovania kódu zo stavových diagramov v UML. Práca zahŕňa analýzu popisu stavových automatov v jazyku UML, nasleduje popis formátu vstupného súboru navrhovaného generátora a návrh generátora samotného. V rámci práce bol generátor kompletne naimplementovaný spolu s ďalšími funkčnými požiadavkami. Popis výslednej implementácie uzatvára túto diplomovú prácu.

Klíčová slova: Generovanie Kódu, Stavový Automat, Jazyk UML

Title: Code Generation from UML State Machine Description
Author: Ľuboš Píš
Department: Department of Software Engineering
Supervisor: doc. Ing. Karel Richta, CSc.
Supervisor's e-mail address: richta@ksi.ms.mff.cuni.cz

Abstract: This paper discusses the implementation of a suitable algorithm for code generation from UML state machine diagrams. The work includes analysis of state machines described in UML, followed by a description of the input file format of the proposed design of the generator and the generator itself. The generator was fully implemented in the work along with other functional requirements. At the end of this thesis is a description of the resulting implementation.

Keywords: Code Generation, State Machine, UML Language

Obsah

1. Zadanie

Pre systematický vývoj kvalitného softwaru je dôležité, aby samotnému programovaniu predchádzal proces analýzy a návrhu systému. Zaužívaným štandardom využívaným v rôznych fázach tohto procesu je jazyk UML. S jeho využitím sa dá dosiahnuť vytvorenie pomerne detailného modelu systému. Problém ale nastáva vo fáze, keď je model pripravený a treba ho pretaviť na kód. Programátor musí začať vpodstate od nuly, podrobne skúmať model a transformovať jeho jednotlivé aspekty do slov programovacieho jazyka. Nedá sa vylúčiť, že pri tomto procese nenastane nejaká chyba a tak aj dobrý návrh môže skončiť ako zlý systém.

Problém je aj s úpravami systému. Aby ostal model konzistentný s programom a dal sa tak využiť k jeho dokumentácii, musí byť každá zmena zanesená najskôr samostatne do modelu a následne do systému. Vymenované aspekty mierne degradujú celý proces návrhu softwaru, pretože je veľmi náročné ho plynule prepojiť s procesom implementácie. Ideálne by preto bolo, aby sa model po fáze návrhu stal priamo kódom. Týmto sa zaoberá oblasť softwarového inžinierstva nazvaná Model-Driven Development (vývoj riadený modelom). Tento prístup zatiaľ nie je tak ďaleko, aby sa jeho prostredníctvom dal vyvíjať komplexný software až do tej miery, že by skutočne nebolo treba ďalšej programátorskej práce, existujú už ale aspoň čiastkové riešenia, ktoré uľahčia programátorovi prácu a zároveň slúžia k udržaniu konzistencie medzi modelom a kódom. Najúspešnejším na tomto poli je generovanie kódu tried z modelov tried, pretože tu je veľmi citeľné a ľahko dešifrovateľné párovanie medzi modelom a kódom (prakticky 1:1).

Existujú ale aj iné modely systému, ktoré by bolo užitočné vedieť transformovať na kód. Napríklad modely, ktoré viac popisujú logiku chovania systému, stavy v akých sa počas chodu nachádza a jeho interakcie s užívateľom či s inými systémami. V jazyku UML sú tieto modely reprezentované napríklad v podobe modelov prípadov použití, aktivít, alebo stavových automatov. V tejto oblasti už riešenia nie sú tak dostupné a rozšírené. Cieľom tejto práce je práve jedno takéto riešenie vytvoriť. Ako vstupný model boli zvolené stavové automaty, ktoré sa používajú k modelovaniu histórie životného cyklu jedného reaktívneho objektu ako konečného stavového automatu.

1.1 Požiadavky zadania

Z pohľadu použiteľnosti riešenia je tu niekoľko hlavných požiadaviek, ktoré musia byť splnené, aby bola úloha úspešne splnená. Nasledujúci list zhŕňa

všetky požiadavky:

1. Aby generovanie nebolo úzko naviazané na jeden konkrétny výstupný programovací jazyk navrhnúť generátor tak, aby podporoval moduly umožňujúce generovanie kódu v rôznych cieľových jazykoch, aby si užívateľ mohol vybrať výstupný jazyk, prípadne si doprogramovať modul pre ďalší výstupný jazyk.

2. Umožniť rôzne formáty vstupných súborov s tým, že formát bude automaticky detekovaný a že užívateľ si bude môcť doprogramovať vlastný parser pre ľubovoľný formát súboru.

3. Ponechať ťažisko práce a logiky na samotnom generátore tak, aby užívateľ, ktorý si chce doprogramovať generovanie pre ďalší jazyk, nemusel programovať logiku generovania kódu zo stavového automatu, ale len dodal modul ktorý bude vedieť vyjadriť základné javy kódu v danom jazyku.

4. Návrh overiť na prototypových moduloch, ktoré budú vedieť spracovávať jeden vstupný formát a následne generovať kostru kódu v jednom z bežne používaných jazykov. Keďže projekt je implementovaný v jazyku C#, ako ukážkový výstupný formát bol zvolený práve tento jazyk. Ako vstupný formát bol zvolený jazyk XMI (vo verzii 2.1), ktorý je štandardom pre prenos modelov v jazyku UML medzi rôznymi programami. Pre jeho rozšírenosť je najpravdepodobnejšie jeho použitie v praxi. Ostatné formáty sú voči nemu len doplnkovými a očakáva sa, že v typickom scenári použitia nebude potrebné implementovať vlastný modul pre vstupný jazyk. Z tohto dôvodu je jazyku XMI venovaná aj samostatná kapitola.

1.2 Štruktúra práce

Štruktúra práce sleduje postup, ako bola práca vypracovaná. Táto úvodná časť sumarizuje celé zadanie, uvádza do problému a predkladá všetky hlavné požiadavky.

Druhá časť analyzuje existujúce riešenia a možnosti ich použitia tak, aby ilustrovala vstupný stav problematiky a pomohla porovnaniu s výsledkom implementácie na konci práce.

Tretia časť popisuje jazyk UML a modelovanie stavových automatov s jeho použitím. Zameriava sa podrobne na jednotlivé aspekty a možnosti stavových automatov, aby poukázala na čo všetko stavové automaty slúžia a aké situácie sa prostredníctvom nich dajú modelovať.

Štvrtá časť popisuje jazyk XMI a reprezentáciu stavových automatov v tomto jazyku. Jej účelom je ukázať ako budú vyzerať vstupné súbory a ako sa jednotlivé aspekty stavových automatov reprezentujú v jazyku XMI.

Piata časť uvádza jednotlivé aspekty zvoleného riešenia, ktoré bolo implementované a voľby, ktoré k nemu viedli vychádzajúc z požiadaviek práce.

Šiesta časť popisuje voľby, ktoré pri implementácii museli byť vykonané vzhľadom k potrebám zvoleného riešenia a podrobne popisuje implementáciu riešenia a jeho použitie.

2. Prehľad existujúcich riešení

Na poli generovania kódu nie je veľmi veľa dostupných nástrojov, ktoré by generovali kód zo stavových automatov. Ešte menej je takých, ktoré by sa na toto generovanie špecializovali. Zväčša ide o nástroje, ktoré sa venujú generovaniu kódu v širšom merítku a generovanie kódu zo stavových automatov je u nich len okrajová funkcionalita, ktorá umožňuje len pridanie nejakých základných dynamických vlastností generovaným statickým triedam. Tieto nástroje sú často ešte ďalej spojené aj s nástrojom pre návrh modelov (obvykle v jazyku UML) a tieto modely sú zároveň aj vstupom pre generovanie. Voľba výstupného jazyka je u väčšiny dostupných nástrojov bežná a podporovaných býva niekoľko najbežnejších programovacích jazykov. Výstupný kód bol vo všetkých prípadoch praktickou implementáciou automatu, to znamená, že išlo interne o automat, ktorý prijímal udalosti a na základe nich menil svoje stavy. Užívateľ generovaného kódu mohol potom tento automat začleniť do svojho kódu a prostredníctvom jeho metód mu posielať udalosti, vyvolávať prechody a zisťovať aktuálny stav automatu.

Nasleduje výčet predstaviteľov dostupných riešení a ich stručný popis:

1. **StateForge** - Ide o kombináciu online nástroja na tvorbu stavových automatov a troch samostatných generátorov pre štyri rôzne výstupné jazyky - C#, VB.NET, Javu a C++. Nástroj na tvorbu stavových automatov umožňuje vytvárať diagramy stavových automatov graficky. Nejde ale o diagramy v jazyku UML, aj keď sa tu nachádzajú určité podobnosti. Výsledné diagramy sú priestorovo úspornejšie, ale menej zrozumiteľné a popisné. Tento editor je dostupný zdarma. Počas vytvárania diagramu je automaticky generované XML, ktoré tento diagram popisuje. Nejde však o formát XMI, ale o vlastný formát. Toto XML môže byť následne použité ako vstup pre jeden z generátorov. Generátory podporujú pomerne širokú škálu javov, ako napríklad hierarchickosť, paralelnosť, asynchrónnosť, vykonanie príkazov pri vstupe do stavu a pri výstupe zo stavu či stavy s históriou. Tieto generátory sú ale spoplatnené - každý z nich stojí 199 Eur. Kvalita výstupného kódu je pomerne vysoká, no kvôli málo intuitívnemu vytváraniu diagramov a výrazne vysokej cene za generátor je výsledný dojem znížený

2. **Altova UModel** - Ide o komerčný nástroj umožňujúci sofistikované vytváranie modelov v jazyku UML a následné generovanie kódu do jazykov Java, C#, a VB.NET. Generovanie je sústredené okolo generovania kódu z diagramov tried, ak však trieda obsahuje stavový automat,

je možné vygenerovať kód aj pre tento automat. Obmedzením je, že každý prechod musí mať definovanú udalosť, ktorá ho vyvoláva a každá udalosť je striktne viazaná na volanie nejakej z metód triedy. Výstupný kód ponúka pomerne vysokú mieru abstrakcie a rozšíriteľnosti - metódy na prácu s automatom sú deklarované ako virtuálne a každý zo stavov je reprezentovaný ako samostatná trieda. Tento prístup sa výrazne opiera o vedomosť, že výstupné jazyky podporujú všetky potrebné vlastnosti. Celý nástroj je dostupný za 299 Eur. Z pohľadu všetkých funkcií, ktoré ponúka, je táto cena adekvátna, v prípade záujmu len o samotné generovanie kódu zo stavových automatov by ale bola pomerne vysoká.

3. **FSMGenerator** - Ide o open source projekt vytvorený v rokoch 2002 až 2003 Pavlom Bekkermanom. Vstupom pre generovanie kódu je deterministický stavový automat zapísaný pomocou špeciálneho konfiguračného súboru. V tomto prípade ide skutočne o konečné stavové automaty tak, ako sú známe z teórie automatov a preto sú možnosti oproti stavovým automatom v uml pomerne redukované. Nie sú podporované napríklad vnorené stavy alebo podmienené prechody. Pozitívom celého projektu je, že umožňuje rozšírenie o nové generátory a užívateľ si tak môže doprogramovať generovanie do vlastného výstupného kódu. Pre tento účel je tu dostupné dobre zdokumentované API. Nevýhodou tohto rozšírenia je, že si vyžaduje porozumenie samotného generovania, teda užívateľ sám musí implementovať priamo prevod automatu na kód. Ako vodítko tu slúžia len zdrojové kódy generátorov pre C++ a Javu a tak je toto rozšírenie použiteľné len pre zručných programátorov.

4. **AutoFSM** - Ide o súčasť voľne dostupného nástroja Autogen, ktorý slúži na automatizované generovanie textu a programov. Podobne ako u programu FSMGenerator je vstupný automat obmedzený na definíciu udalostí, stavov a prechodov medzi udalosťami. Vo vygenerovanom kóde je zadefinovaná prechodová tabuľka a o vykonávanie automatu sa stará vygenerovaný veľký switch, ktorý obsluhuje jednotlivé prechody. Kód je generovaný do jazyka C alebo C++, je ale možné doprogramovať ďalšie šablóny na generovanie do ďalších výstupných jazykov.

5. **SinelaboreRT** - Ide o spoplatnený software zameraný len priamo na generovanie kódu zo stavových automatov. Vstupom sú automaty definované prostredníctvom jazyka UML. Podporované sú takmer všetky dôležité javy stavových automatov okrem paralelných podautomatov. Generovaný kód je dobre čitateľný a automat je opäť reprezentovaný ako veľký switch cez všetky možné stavy. Má v sebe zabudovanú širokú podporu pre rôzne vstupné formáty (Cadifra, Enterprise Architect,

Magic Draw a ďalšie) ako aj pre tri výstupné jazyky Java, C# a C++. SinelaboreRT je spoplatnený sumou 136 Eur, čo je vzhľadom jeho možnostiam a kvalite výstupného kódu pomerne adekvátna cena.

3. Jazyk UML a stavový model

Keď bol v roku 1997 predstavený Unified Modeling Language (UML), stal sa rýchlo akceptovaným v oblasti softwarového priemyslu, ako štandardný grafický jazyk pre špecifikovanie, konštruovanie, vizualizáciu a dokumentáciu náročných softwarových systémov. UML poskytuje každému, kto je zainteresovaný do produkcie, vývoja a údržby softvéru, štandardné značenie pre vyjadrenie návrhu systému. UML poskytuje spôsob ako modelovať "nielen štruktúru aplikácie, chovanie a architektúru, ale aj biznis proces a štruktúru údajov." (3).

Stavový automat slúži na modelovanie správania jedného objektu. Popisuje správanie, ktoré je špecifikované postupnosťou stavov, ktorými prechádza objekt počas svojho životného cyklu v reakcii na udalosti a jeho reakciami na tieto udalosti.

Stavové automaty sa používajú na modelovanie dynamických aspektov systému, čo zahŕňa špecifikovanie životného cyklu inštancií tried, prípadov použitia (use case), či celého systému. Tieto inštancie môžu odpovedať na udalosti, ako sú signály, operácie, alebo plynutie času. Keď nastane udalosť, v závislosti od aktuálneho stavu objektu nastane nejaká aktivita. Aktivita je prebiehajúci, neatomický proces v rámci stavového automatu. Výsledkom aktivít je napokon nejaká akcia, ktorá pozostáva z atomických výpočtov, ktorých výsledkom je zmena stavu modelu. Stav objektu je situácia v životnom cykle objektu, počas ktorej spĺňa nejakú podmienku, vykonáva nejakú aktivitu, alebo čaká na nejakú udalosť.

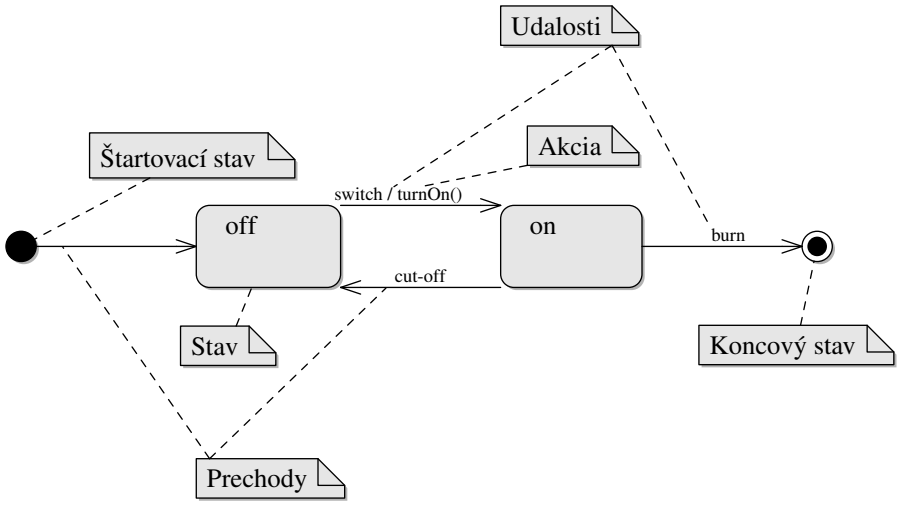
Stavový automat sa dá vizualizovať pomocou vytýčenia možných stavov objektov a prechodov medzi týmito stavmi. Dobre štrukturované stavové automaty sú ako dobre štrukturované algoritmy: Sú efektívne, jednoduché, prispôsobiteľné a zrozumiteľné.

3.1 Úvod

Mnoho náročných softwarových systémov sa chová tak, že riadiaci objekt vykonáva činnosť nepretržite, ale prispôsobuje sa zmenám a aktivitám.

V UML sa statické aspekty systému modelujú pomocou elementov, ako sú diagramy tried a objektov. Tieto diagramy umožňujú vizualizovať, špecifikovať, konštruovať a dokumentovať záležitosti, ktoré žijú v systéme - triedy, rozhrania, komponenty, uzly a prípady použitia a ich inštancie, spolu s popisom vzájomných vzťahov týchto vecí.

Dynamické aspekty systému sa popisujú v UML pomocou stavových auto-



Obrázek 3.1: Stavový diagram

matov. Stavový automat modeluje životný cyklus jedného objektu, či už je to inštancia triedy, prípad použitia, alebo dokonca celý systém. Objekt môže byť počas doby svojej existencie vystavený rôznym udalostiam, ako napríklad signál, vyvolanie operácie, vytvorenie či zrušenie objektu, plynutie času, alebo zmena nejakej podmienky. Na tieto udalosti, objekt reaguje nejakou akciou, ktorá reprezentuje atomický výpočet, ktorého výsledkom je zmena stavu objektu. Správanie takéhoto objektu je preto ovplyvnené minulosťou. Objekt môže obdržať udalosť, odpovedať akciou a potom zmeniť svoj stav. Objekt môže obdržať inú udalosť a jeho odpoveď môže byť odlišná v závislosti na jeho aktuálnom stave v odpovedi na predchádzajúcu udalosť.

Stavový automat slúži k modelovaniu správania sa akéhokoľvek modelového elementu, hoci najčastejšie je ním trieda, prípad použitia, alebo celý systém.

UML ponúka grafickú reprezentáciu stavov, prechodov, udalostí a akcií, ako ukazuje obrázok 3.1. Toto značenie umožňuje vizualizovať správanie objektu spôsobom, ktorý umožní zvýrazniť dôležité prvky života tohto objektu.

3.2 Pojmy a koncepcie

Stavový automat môžme definovať ako chovanie, ktoré špecifikuje platné sekvencie aktivít, ktorými prechádza objekt, alebo systém počas svojho života

v odpovedi na udalosti spolu s odpoveďami a akciami. *Stav* je situácia počas doby existencie objektu, kedy môže objekt spracovávať udalosti. Stav má meno a môže mať vstupnú akciu a výstupnú akciu. *Udalosť* aktivuje prechod. Nazýva sa tiež aktivátor prechodu. *Prechod* prevádza objekt z jedného stavu do druhého. Prechod je obvykle vyvolaný udalosťou a jeho vykonávanie obvykle sprevádza séria akcií. *Aktivita* je prebiehehajúci neatomický výpočet v rámci stavového automatu. *Akcia* je vykonateľný atomický výpočet, ktorého výsledkom je zmena stavu modelu.

Graficky je stav zobrazený ako obdĺžnik s oblými rohmi. Prechod je zobrazený ako plná orientovaná čiara.

3.2.1 Kontext

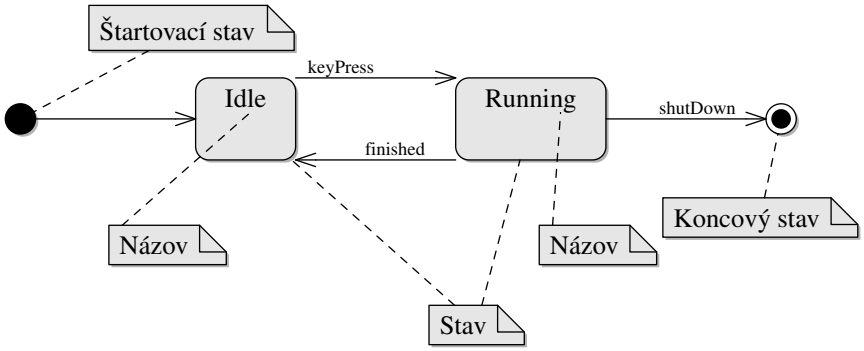
Každý objekt má životný cyklus. Pri vytvorení sa objekt rodí, pri deštrukcii objekt končí. Medzitým môže objekt pôsobiť na iné objekty (tým, že im posiela správy), tak ako môže byť pôsobené naň (tým, že je cieľom správy). V mnohých prípadoch tieto správy budú jednoduché synchrónne volania operácií. Napríklad objekt `Customer` môže vyvolať operáciu `getAccountBalance` na inštancii triedy `BankAccount`. Objekty ako tieto, nepotrebujú stavový automat na popis ich správania, pretože ich súčasné správanie nezávisí na ich minulosti.

V iných typoch systémov sa dajú pozorovať objekty, ktoré musia odpovedať na signály, ktoré sú asynchrónnymi stimulmi vymieňanými medzi inštanciami. Napríklad mobilný telefón musí odpovedať na náhodné telefonické hovory (z iných telefónov), udalosti klávesnice (od užívateľa iniciujúceho telefonický hovor) a na udalosti zo siete (keď telefón prechádza z jednej bunky do inej). Podobne je možné pozorovať objekty, ktorých aktuálne správanie závisí na ich minulom správaní. Napríklad lietadlo môže letieť len ak predtým štartovalo a je zrejmé, že niekedy v budúcnosti by zas malo pristáť.

Správanie objektov, ktoré musia odpovedať na asynchrónne podnety, alebo ktorých správanie závisí na ich minulosti, je najlepšie popísateľné použitím stavového automatu. Stavové automaty sa taktiež používajú na modelovanie správania celých systémov, obzvlášť reaktívnych systémov, ktoré musia odpovedať na signály všetkých činiteľov zvonku systému.

3.2.2 Stavy

Stav je situácia v živote objektu, počas ktorej spĺňa nejakú podmienku, vykonáva nejakú aktivitu, alebo čaká na nejakú udalosť. Objekt zotrváva v stave na spravidla konečne dlhú dobu. Napríklad `Ohrievač` v dome môže byť v jednom zo štyroch stavov: `Idle` (čakanie na príkaz k začatiu vyhrievania domu),



Obrázek 3.2: Stavy

`Activating` (je zapnutý, ale čaká na dosiahnutie teploty), `Active` (je zapnutý a vyhrieva dom) a `ShuttingDown` (je vypnutý, vyháňa sa zostatkové teplo zo systému).

Keď je stavový automat objektu v danom stave, hovoríme, že objekt je v danom stave. Napríklad inštancia `Ohrievač` môže byť v stave `Idle`, alebo napríklad v stave `ShuttingDown`.

Stav sa skladá z niekoľkých súčastí.

1. Názov - Textový reťazec, ktorý odlišuje stav od iných stavov.

2. Vstupné/výstupné akcie - Akcie vykonané jednotlivo pri vstupe do stavu a výstupe z neho.

3. Interné prechody - Prechody, ktoré sú ošetrené bez spôsobenia zmeny stavu. Je možné ich nasimulovať aj prostredníctvom normálnych (externých) prechodov za cenu skomplikovania automatu. Ide teda len o výrazovú skratku.

4. Podautomaty - Vnorená štruktúra stavu obsahujúca nezávislé (postupne aktívne), alebo paralelné (súbežne aktívne) podstavy.

5. Asynchrónne udalosti - List udalostí, ktoré môžu v automate nastať a ktoré nie sú ošetrené v tomto stave, ale namiesto toho sú odložené pre ošetrenie v inom stave.

Ako ukazuje obrázok 3.2, stav sa reprezentuje ako obdĺžnik s oblými rohmi.

Štartovací a koncový stav - Ako ukazuje obrázok, sú dva špeciálne stavy, ktoré môžu byť definované pre stavový automat objektu. Prvým

je štartovací stav, ktorý označuje implicitné štartovacie miesto pre stavový automat, alebo podautomat. Štartovací stav je reprezentovaný ako vyplnený čierny kruh. Druhým je koncový stav, ktorý označuje vykonanie stavového automatu, alebo uzavretie stavu, ktorý bol dokončený. Koncový stav je reprezentovaný ako vyplnený čierny kruh obklopený nevyplneným kruhom.

3.2.3 Prechody

Prechod je vzťah medzi dvoma stavmi, vyjadrujúci, že objekt v prvom stave vykoná určité akcie a vstúpi do druhého stavu, keď nastane určitá udalosť a sú splnené určité podmienky. Pri takejto zmene stavu hovoríme, že došlo k prechodu. Kým dôjde k prechodu hovoríme, že objekt je v zdrojovom stave, potom ako nastane prechod hovoríme, že je v cieľovom stave. Napríklad `Ohrievač` môže prechádzať zo stavu `Idle` do stavu `Activating`, keď nastane udalosť `tooCold`.

Prechod sa skladá z piatich súčastí.

1. Zdrojový stav - Stav ovplyvnený prechodom. Ak je objekt v zdrojovom stave, vychádzajúci prechod môže nastať, keď objekt obdrží udalosť vyvolávajúcu prechod a ak je splnená konrolná podmienka (ak je nejaká definovaná).

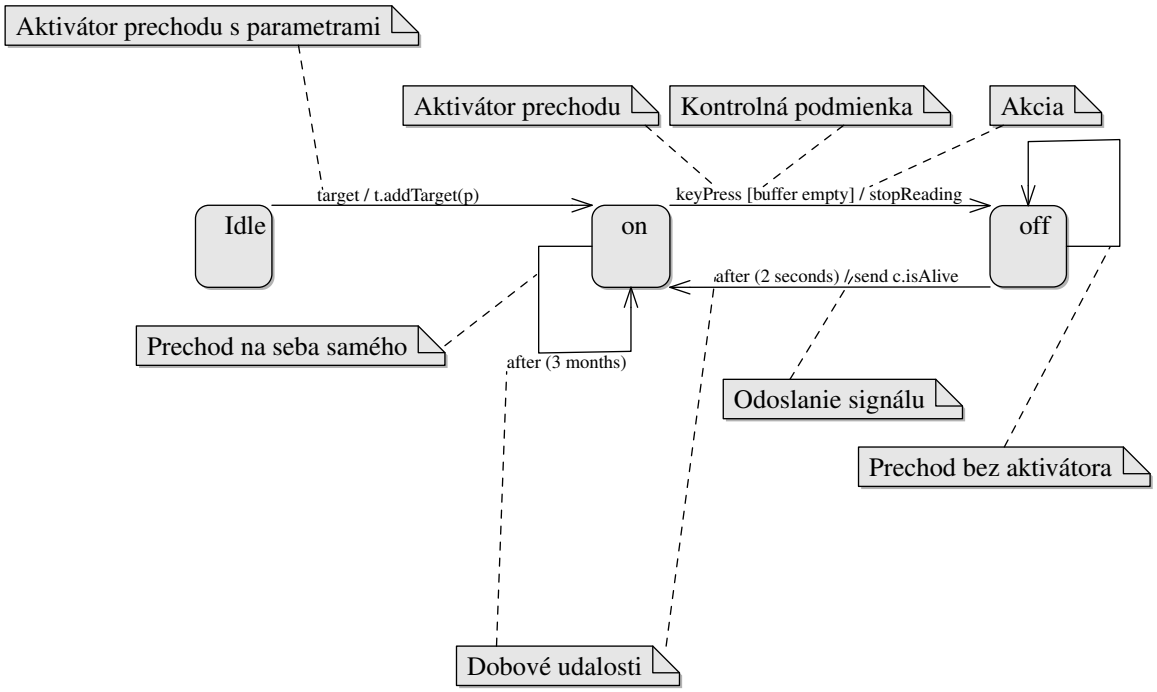
2. Aktivátor prechodu - Udalosť, ktorej vyskytnutie sa u objektu v zdrojovom stave spôsobí, že môže nastať prechod, ak je splnená jeho kontrolná podmienka.

3. Kontrolná podmienka - podmienka, alebo množina podmienok spojená s aktivátorom prechodu je vyhodnocovaná vždy keď je vyhodnocovaná aktivačná udalosť. Ak je výraz vyhodnotený ako True, môže dôjsť k prechodu. Ak je výraz vyhodnotený ako False, k prechodu nedôjde a ak neexistuje iný prechod, ktorý by mohol byť spustený tou istou udalosťou, udalosť sa nespracuje.

4. Akcia - Vykonateľný atomický výpočet, ktorý môže priamo pôsobiť na objekt, ku ktorému patrí stavový automat a nepriamo na objekty, ktoré sú pre objekt viditeľné.

5. Cieľový stav - Stav, ktorý sa stane aktívnym po dokončení prechodu.

Ako ukazuje obrázok 3.3, prechod sa zobrazuje ako plná orientovaná čiara zo zdrojového do cieľového stavu. Prechod na seba samého je prechod, ktorého zdrojový a cieľový stav sú totožné.



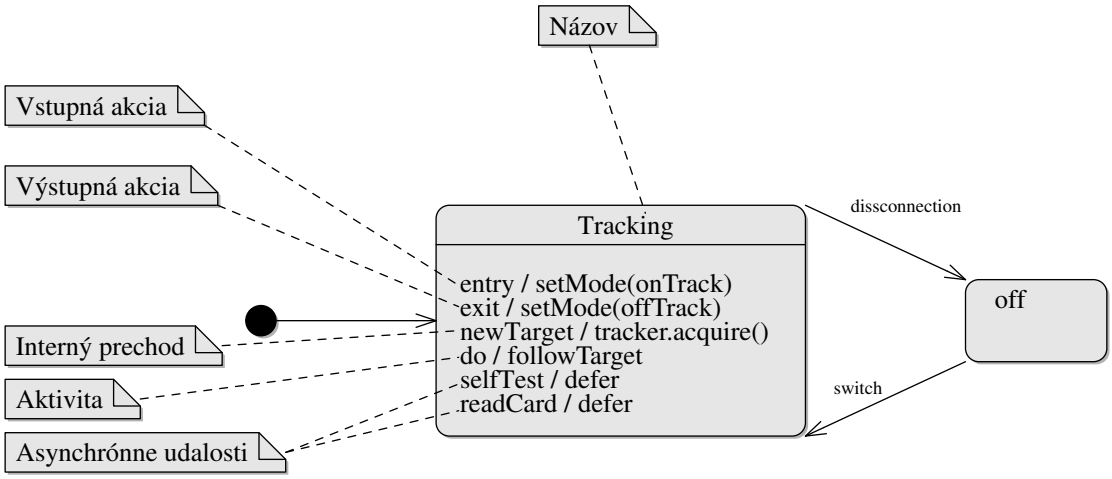
Obrázek 3.3: Prechody

Aktivátor prechodu - Udalosť je popis významného javu, ktorý má pozíciu v čase a priestore. Z pohľadu stavových automatov je udalosť výskyt podnetu, ktorý môže aktivovať stavový prechod. Ako je ukázané na predošlom obrázku, udalosti môžu obsahovať signály, volania, chod času, alebo zmenu stavu.

Je tiež možné mať prechod bez aktivátora, reprezentovaný prechodom bez aktivátora prechodu. Prechod bez aktivátora, tiež volaný prechod po ukončení, je aktivovaný implicitne, keď jeho zdrojový stav ukončil svoju aktivitu.

Kontrolná podmienka - Ako ukazuje predošlý obrázok, kontrolná podmienka sa zobrazuje ako Booleovský výraz uzavretý v hranatých zátvorkách a umiestnený za aktivátorom prechodu. Kontrolná podmienka je vyhodnotená len po výskyte aktivátora prechodu pre jej prechod. Preto je možné mať viacero prechodov z jedného zdroja a s tým istým aktivátorom prechodu, pokiaľ sa tieto podmienky neprekrývajú.

Kontrolná podmienka je vyhodnotená len raz pre každý prechod keď nastane udalosť, ale môže byť vyhodnotená opäť ak je prechod znova aktivovaný. Do booleovského výrazu je možné zahrnúť podmienky o stave objektu (napríklad výraz `aHeater in Idle`, ktorý sa vyhodnotí ako True ak je objekt `aHeater` práve v stave `Idle`).



Obrázek 3.4: Pokročilé stavy a prechody

Akcia - Akcia je spustiteľný atomický proces. Akcie môžu obsahovať volania operácií (objektu, ku ktorému patrí stavový automat, rovnako ako iných viditeľných objektov), vytvorenie, alebo deštrukciu iného objektu, alebo zaslanie signálu objektu.

Akcia je atomická, čo znamená, že nemôže byť prerušená udalosťou a preto beží až do konca. Toto je v kontraste k aktivite, ktorá môže byť prerušená inými udalosťami.

3.2.4 Pokročilé stavy a prechody

S použitím základných možností stavov a prechodov je možné v UML modelovať širokú škálu chovania. Využitie týchto možností vedie k plochým stavovým automatom, čo znamená že modely správania nebudú pozostávať z ničoho iného ako z čiar (prechody) a vrcholov (stavy).

Stavové automaty v UML majú ale mnoho pokročilých možností, ktoré pomáhajú vytvoriť komplexné modely správania. Tieto možnosti často redukujú počet potrebných stavov a prechodov a stanovujú mnoho všeobecných a v určitom smere komplexných postupov, ktoré by sa inak museli zložito definovať s využitím plochých stavových automatov. Tieto možnosti zahŕňajú vstupné a výstupné akcie, interné prechody, aktivity a asynchrónne udalosti.

Vstupné a výstupné akcie - V mnohých modelových situáciách, je potrebné vykonať tú istú akciu pri každom vstupe do stavu bez ohľadu na to, ktorý prechod do neho bol vykonaný. Podobne pri opustení stavu je potrebné vykonať tú istú akciu bez ohľadu na to, ktorý prechod z neho bol vykonaný. Napríklad v riadiacom systéme rakety môže byť potrebné explicitne ozná-

miť, že systém je `onTrack` vždy keď sa vstúpi do stavu `Tracking` a `offTrack` vždy keď sa z tohto stavu vystúpi. S použitím plochých stavových automatov sa dá dosiahnuť tento efekt vložením patričných akcií na každý vstupný a výstupný prechod. Toto je ale mierne náchylné k chybám - je nutné si pamätať, že treba pridať tieto akcie vždy keď sa pridáva nový prechod. Navyše modifikovanie tejto akcie znamená, že sa musia obmeniť všetky okolité prechody.

Ako ukazuje obrázok 3.4, UML poskytuje skratku pre tento postup. V symbole pre stav, sa môže zahrnúť vstupná akcia (označená kľúčovým slovom `entry`) a výstupná akcia (označená kľúčovým slovom `exit`), spolu s príslušnou akciou. Vždy pri vstupe do stavu je vykonaná vstupná akcia a vždy pri výstupe zo stavu je vykonaná výstupná akcia.

Interné prechody - Niekedy v rámci stavu môže nastať udalosť, ktorú je potrebné spracovať bez opustenia stavu. Tomuto javu sa hovorí interné prechody a sú nepatrne odlišné od prechodov na seba samého. V prechodoch na seba samého, ako vidno na obrázku 3.3, udalosť aktivuje prechod, opustí sa stav, je vykonaná akcia (ak je nejaká definovaná) a potom sa znova vstúpi do toho istého stavu. Keďže tento prechod opustí stav a potom vstúpi do stavu, prechod na seba samého vyvolá výstupnú akciu stavu, potom vyvolá akciu prechodu do seba samého a nakoniec vyvolá vstupnú akciu stavu. Môže sa ale stať, že je potrebné obslúžiť udalosť, ale bez spustenia vstupnej a výstupnej akcie stavu. S použitím plochých stavových automatov sa tento efekt dá dosiahnuť, ale je potrebné si dôsledne pamätať, ktorý zo stavových prechodov má vstupné a výstupné akcie a ktorý nie.

Ako ukazuje obrázok 3.4, UML poskytuje skratku i pre tento postup (napríklad pre udalosť `newTarget`). V symbole pre stav je možné zahrnúť interný prechod (označený udalosťou). Vždy, keď je vnútri stavu aktivovaná táto udalosť, vykoná sa príslušná akcia bez opustenia stavu a opätovného vstupu do neho. Preto je udalosť obslúžená bez vyvolania vstupnej a výstupnej akcie stavu.

Aktivity - Keď je objekt v stave, obvykle zotrváva bez práce a čaká na výskyt udalosti. Niekedy je ale potrebné modelovať prebiehajúcu aktivitu. Kým je objekt v stave, vykonáva nejakú prácu, ktorá bude pokračovať, kým nebude prerušená udalosťou. Napríklad kým je objekt v stave `Tracking`, môže vykonávať aktivitu `followTarget`. Ako ukazuje obrázok 3.4, v UML sa používa špeciálny prechod `do` na označenie práce, ktorá sa bude vykonávať v stave po tom, ako je vykonaná vstupná akcia. Je možné taktiež špecifikovať postupnosť akcií - napríklad, `do / op1(a); op2(b); op3(c)`. Akcie nie sú nikdy prerušiteľné, ale postupnosti akcií sú. Medzi každými dvoma akciami (oddelenými bodkočiarkami), môžu byť nadradeným stavom ošetrené udalosti, čoho výsledkom je prechod mimo stav.

Asynchrónne udalosti - V obrázku 3.3 je len jeden prechod vedúci von zo stavu `Idle`, aktivovaný udalosťou `target`. V stave `Idle` budú všetky udalosti okrem `target` a okrem tých, ktoré sú ošetrené jeho podstavmi, stratené. To znamená, že udalosť môže nastať, ale bude ignorovaná a výsledkom jej prítomnosti nebude žiadna akcia.

V každej modelovej situácii je potrebné zachytiť niektoré udalosti a ignorovať iné. Tie, ktoré je potrebné zachytiť sa zahrnú ako aktivátory prechodov. Tie, ktoré sa majú ignorovať sa jednoducho vynechajú. V niektorých modelových situáciách je ale potrebné zachytiť niektoré udalosti, ale odložiť reakciu na ne na neskôr. Napríklad v stave `Tracking` môže byť potrebné odložiť odpoveď na signály ako `selfTest`, pravdepodobne poslané nejakou službou údržby systému.

V UML je možné popísať toto chovanie použitím asynchrónnych udalostí. Asynchrónna udalosť je zoznam udalostí, ktorých výskyt v stave je odložený kým nebude aktívny stav, v ktorom nie sú tieto udalosti asynchrónne a potom nastanú a môžu aktivovať prechody ako keby práve nastali. Ako je vidno na predošlom obrázku, je možné zadať asynchrónnu udalosť zapísaním udalosti so špeciálnou akciou `defer`. V tomto príklade môžu v stave `Tracking` nastať udalosti `selfTest`, ale budú pozdržané kým bude objekt v stave `off`, kedy sa zjavia, ako by len práve vtedy nastali.

3.2.5 Podstavy

Pokročilé možnosti stavov a prechodov, ktoré sme popísali, riešia mnoho obvyklých problémov modelovania stavových automatov. Je tu ale ešte jedna možnosť UML stavových automatov - podstavy, ktorá pomáha ešte viac zjednodušiť modelovanie komplexného správania. Podstav je stav, ktorý je vnorený do iného. Napríklad objekt reprezentujúci ohrievač môže byť v stave `Heating` a v rámci tohto stavu `Heating`, môže byť vo vnorenom stave nazvanom `Activating`. V tomto prípade je správne hovoriť, že objekt je v oboch stavoch `Heating` aj `Activating`

Jednoduchý stav je stav, ktorý nemá podštruktúru. Stav, ktorý má podstavy - teda vnorené stavy, sa nazýva skladaný stav. Skladaný stav môže obsahovať buď simultánne, alebo postupné (nezávislé) podstavy. V UML sa zobrazuje skladaný stav rovnako ako jednoduchý stav, ale so zvláštnym grafickým priestorom, ktorý zobrazuje vnorený stavový automat. Podstavy môžu byť vnorené do ľubovoľnej hĺbky.

Postupné podstavy - Skladaný stav, ktorý obsahuje jeden vnorený stavový automat, je označovaný ako postupný skladaný stav. Každý podstav dedí všetky prechody od svojho predka.

Použitie nadstavov a podstavov môže stavové diagramy veľmi zjednodušiť.

Ako ukazuje obrázok 3.5 s využitím postupných podstavov existuje jednoduchší spôsob modelovania takýchto problémov. Stav `ATM` má podštruktúru obsahujúcu podstavy `Validating`, `Selecting`, `Processing` a `Printing`. Pri vstupe do stavu `ATM` je vykonaná vstupná akcia `readCard`. Z iniciálneho stavu subštruktúry prejde riadenie na stav `Validating`, potom do stavu `Selecting` a potom do stavu `Processing`. Po stave `Processing` sa môže riadenie vrátiť do stavu `Selecting` (ak zákazník zvolil ďalšiu transakciu), alebo môže prejsť na `Printing`. Po stave `Printing` nastane prechod bez aktivátora do stavu `Shop`. Stav `ATM` má výstupnú akciu `ejectCard`, ktorá vysunie kreditnú kartu zákazníka.

Prechod zo stavu `ATM` do stavu `Shop` je aktivovaný udalosťou `balance > 0`. V každom podstave `ATM`, môže zostatok stúpnuť nad nulu a to vráti automat do stavu `Shop` (ale len po vysunutí zákazníkovej kreditnej karty, čo je akcia vyvolaná pri opúšťaní stavu `ATM` bez ohľadu na to, čo spôsobilo prechod zo stavu). Bez podstavov by bol potrebný prechod aktivovaný `balance > 0` na každom stave podštruktúry.

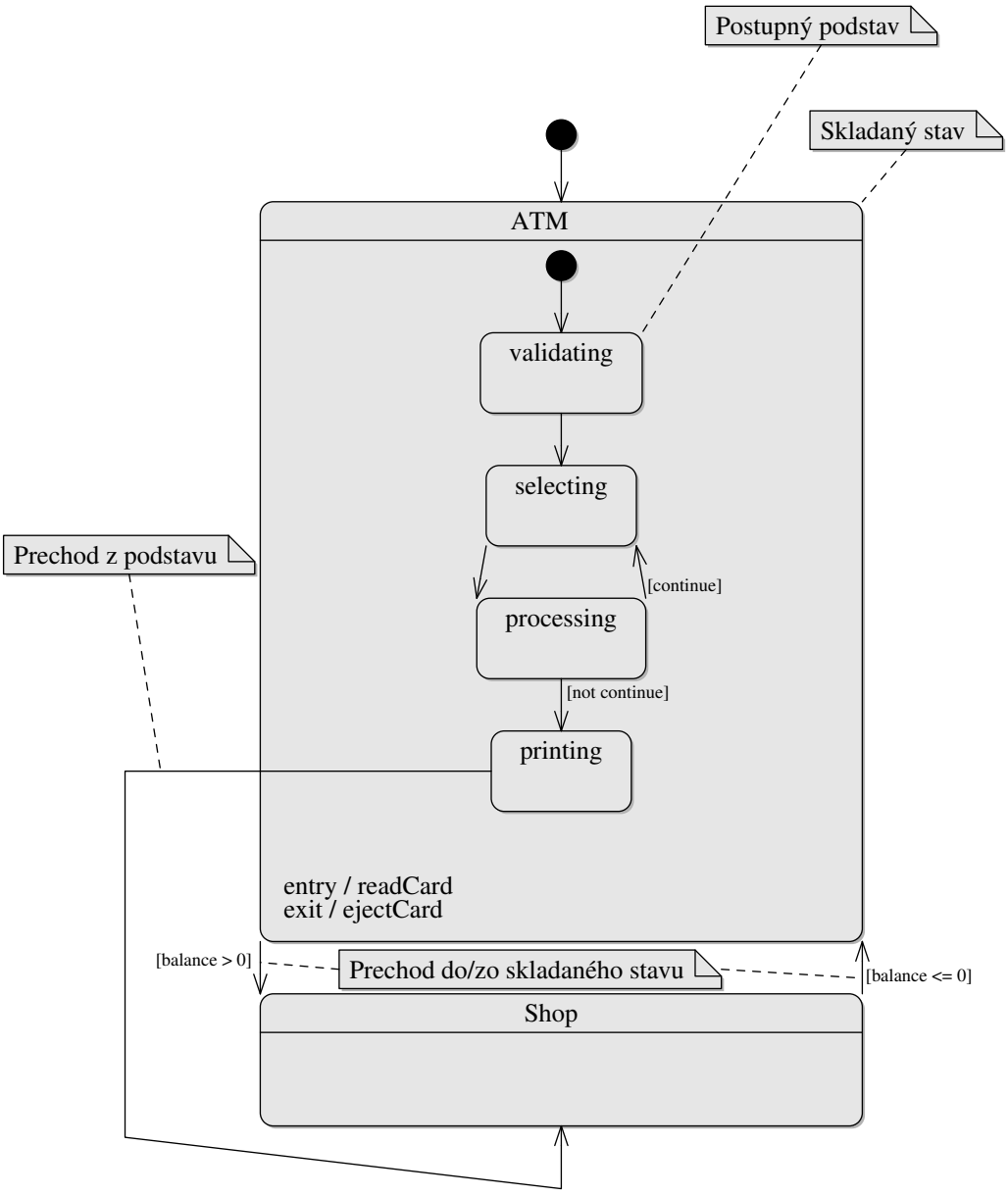
Podstavy ako `Validating` a `Processing` sa nazývajú postupné, alebo sekvenčné podstavy. Pre množinu sekvenčných podstavov z pohľadu nadradeného skladaného stavu sa hovorí, že objekt je v skladanom stave a iba v jednom z daných podstavov (alebo koncovom stave) naraz. Preto postupné podstavy rozdeľujú priestor skladaného stavu na oddelené stavy.

Zo zdroja mimo nadradeného skladaného stavu môže prechod smerovať do skladaného stavu, alebo do podstavu. Ak je cieľom skladaný stav, vnorený stavový automat musí obsahovať štartovací stav, na ktorý prejde riadenie po vstupe do skladaného stavu a po vykonaní vstupnej akcie (ak je definovaná). Ak je cieľom vnorený stav, riadenie prejde na vnorený stav po vykonaní vstupnej akcie (ak je definovaná) skladaného stavu a potom vstupnej akcie (ak je definovaná) podstavu.

Prechod vedúci zo skladaného stavu môže smerovať zo skladaného stavu, alebo podstavu. V oboch prípadoch riadenie najskôr opustí vnorený stav (a ak je definovaná výstupná akcia, tak je vykonaná), potom opustí skladaný stav (a ak je definovaná výstupná akcia, tak je vykonaná). Prechod, ktorého zdrojom je skladaný stav jednoducho odreže (preruší) aktivitu vnoreného stavového automatu.

Stavy s históriou - Stavový automat popisuje dynamické aspekty objektu, ktorého aktuálne správanie závisí na jeho histórii. Stavový automat vpodstate definuje správne usporiadanie stavov, ktorými má objekt prejsť počas svojho životného cyklu.

Pri každom vstupe do skladaného podstavu vždy začína akcia vnoreného stavového automatu na jeho štartovacom stave (ak teda samozrejme prechod



Obrázek 3.5: Postupné podstavy

nevstupuje priamo do podstavu). Sú ale prípady, keď je vhodnejšie modelovať objekt tak, aby si pamätal posledný podstav, ktorý bol aktívny pred opustením skladaného stavu. Napríklad pri modelovaní chovania služby, ktorá vykonáva samostatné zálohovanie počítačov v sieti je vhodné si pamätať, kde sa nachádzal proces vždy, keď bol prerušený napríklad požiadavkou operátora.

S použitím plochých stavových automatov je takéto chovanie možné modelovať, ale je to komplikované. Pre každý postupný podstav by bolo potrebné, aby jeho výstupná akcia odoslala hodnotu do nejakej lokálnej premennej skladaného stavu. Potom by štartovací stav tohto skladaného stavu musel obsahovať prechody do každého podstavu s kontrolnou podmienkou, overujúcou premennú. Takýmto spôsobom by opustenie skladaného stavu spôsobilo, že by bol zapamätaný posledný podstav a vstup do skladaného stavu by prešiel do príslušného podstavu. Takéto riešenie je pomerne komplikované, lebo vyžaduje ošetrenie každého podstavu a nastavenie príslušnej výstupnej akcie. Vzniklo by enormné množstvo prechodov rozvetvujúcich sa z toho istého štartovacieho stavu na rôzne cieľové podstavy s veľmi podobnými (ale rozdielnymi) kontrolnými podmienkami.

Jednoduchší spôsob, ako modelovať tento problém v UML je použitie stavov s históriou. Stav s históriou umožňuje skladanému stavu, ktorý obsahuje postupné podstavy, pamätať si posledný podstav, ktorý bol aktívny pred prechodom zo skladaného stavu. Stav s plytkou históriou je reprezentovaný malým kruhom obsahujúcim symbol `H`.

Ak má prechod aktivovať posledný podstav, vytvorí sa prechod zvonka skladaného stavu priamo do stavu s históriou. Pri prvom vstupe do skladaného stavu stav nemá históriu. Na tento účel má stav s históriou jediný prechod. Cieľ tohoto prechodu udáva stav vnoreného stavového automatu do ktorého sa má vstúpiť pri prvom vstupe do neho.

Keď vnorený stavový automat dosiahne koncový stav, stráca uloženú históriu a správa sa, ako keby sa do neho ešte nikdy nevstúpilo.

Simultánne podstavy - Postupné podstavy sú najbežnejším druhom vnoreného stavového automatu. V určitých modelových situáciách je ale potrebné zadefinovať simultánne podstavy. Tieto podstavy umožňujú definovať dva, alebo viaceré stavové automaty, ktoré pracujú paralelne v kontexte nadradeného objektu.

Napríklad obrázok 3.6 ukazuje ako je stav `Maintenance` rozdelený do dvoch simultánnych podstavov `Testing` a `Commanding`, čo je zobrazené ich vnorením do stavu `Maintenance`, ale oddelením jedného od druhého prerušovanou čiarou. Každý z týchto simultánnych podstavov je ďalej rozdelený na postupné podstavy. Keď kontrola prejde zo stavu `Monitoring` do stavu `Maintenance`,

rozdelí sa na dva simultánne prúdy - nadradený objekt bude v stave `Testing` a zároveň v stave `Commanding`. Okrem toho počas stavu `Commanding` bude nadradený objekt v stave `Waiting` alebo `Command`.

Ako uvádzajú Booch, Rumbaugh a Jacobson (1) Vykonávanie týchto dvoch simultánnych podstavov prebieha paralelne. Každý vnorený stavový automat môže dosiahnuť svoj koncový stav. Ak jeden simultánny podstav dosiahne svoj koncový stav skôr, kontrola v tomto podstave počká na koncový stav druhého z nich. Keď oba vnorené stavové automaty dosiahnu svoje koncové stavy, kontrola z dvoch simultánnych podstavov sa spojí späť do jedného prúdu.

Vždy keď nastane prechod do skladaného stavu rozdeleného na simultánne podstavy, kontrola sa rozdelí do toľkých simultánnych prúdov, koľko je simultánnych podstavov. Podobne, vždy keď nastane prechod zo skladaného podstavu rozdeleného na simultánne podstavy, kontrola sa spojí späť do jedného prúdu. Toto platí v prípade ak všetky simultánne podstavy dosiahnu svoj koncový stav, alebo ak nastane explicitný prechod von z nadradeného skladaného stavu, kontrola sa spojí späť do jedného prúdu.

3.3 Použitie stavových automatov

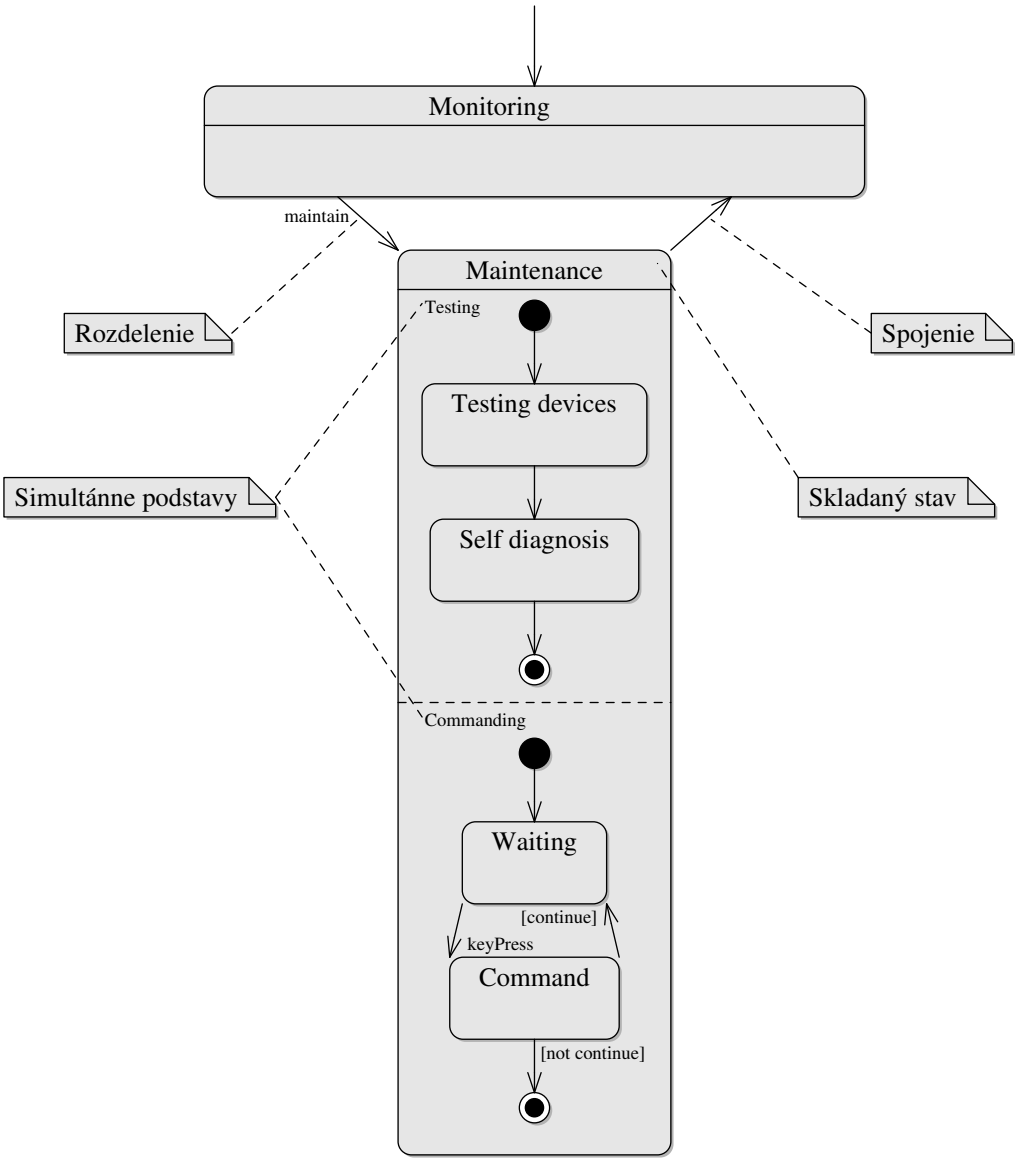
Najčastejším účelom, na ktorý sa používajú stavové automaty, je modelovanie životného cyklu objektu, špeciálne inštancií tried, prípadov použitia a systému ako celku. Stavový automat modeluje správanie jedného samostatného objektu počas jeho životného cyklu. Typické príklady vhodného použitia stavových automatov je možné nájsť pri užívateľských rozhraniach, kontroleroch a zariadeniach.

3.3.1 Modelovanie životného cyklu objektu

Keď sa modeluje životný cyklus objektu, je podstatné definovať tri veci: udalosti na ktoré objekt môže odpovedať, odpoveď na tieto udalosti a dopad minulosti na terajšie správanie. Modelovanie životného cyklu objektu zahŕňa tiež rozhodovanie o poradí, v ktorom objekt môže zmysluplne odpovedať na udalosti, odvtedy ako je objekt vytvorený až kým nie je zrušený.

Na modelovanie životného cyklu objektu treba

- Sformulovať kontext stavového automatu - či je to trieda, prípad použitia, alebo systém ako celok. Ak je kontextom systém ako celok, treba



Obrázek 3.6: Simultánne podstavy

sa zamerať na jedno správanie systému. Teoreticky, každý objekt v systéme môže byť činiteľom v modeli životného cyklu systému a s výnimkou najtriviálnejších systémov môže byť kompletný model veľmi komplikovaný.

- Stanoviť štartovacie a koncové stavy pre objekt. Na usmernenie zvyšku modelu patrične stanoviť možné vstupné a výstupné podmienky štartovacích a koncových stavov.
- Rozhodnúť na aké udalosti môže tento objekt odpovedať. Ak už boli definované, je možné ich nájsť v rozhraniach objektu. Ak ešte neboli definované, treba zvážiť, ktoré objekty môžu pôsobiť na objekt v danom kontexte a aké udalosti môžu posielať.
- Od štartovacieho stavu po koncový umiestniť stavy najvyššej úrovne, ktoré môže objekt nadobúdať. Tieto stavy treba spojiť prechodmi spúšťanými príslušnými udalosťami.
- Pridať k týmto prechodom akcie požadované akcie, ktoré sa majú vykonať vždy spolu s daným prechodom.
- Identifikovať všetky vstupné, alebo výstupné akcie stavov.
- Rozvinúť stavy použitím podstavov, ak je to potrebné.
- Skontrolovať, že všetky udalosti uvedené v stavovom automate súhlasia s udalosťami očakávanými rozhraním objektu. Podobne, treba skontrolovať že všetky udalosti očakávané rozhraním objektu sú ošetrené stavovým automatom.
- Prejsť stavovým automatom buď manuálne, alebo s využitím nástrojov a vyskúšať ho pre očakávané postupnosti udalostí a ich odpovedí. Pozornosť treba venovať najmä hľadaniu nedosiahnuteľných stavov a stavov, v ktorých by automat mohol uviaznuť.
- Po preusporiadaní stavového automatu, ho treba opäť vyskúšať pre očakávané postupnosti a uistiť sa, že sa nezmenila sémantika objektu.

3.4 Dobre štruktúrovaný stavový automat

Pri modelovaní stavových automatov v UML treba mať na pamäti, že každý stavový automat reprezentuje dynamické aspekty jedného samostatného objektu, typicky reprezentuje inštanciu triedy, prípad použitia, alebo systém ako celok. Dobre štruktúrovaný stavový automat

- Je jednoduchý a preto by nemal obsahovať žiadne nadbytočné stavy, alebo prechody.
- Má jasný kontext.
- Je účelný a preto môže splniť svoje chovanie s optimálnym vyvážením času a prostriedkov, ktoré sú vyžadované akciami, ktoré vyvoláva.
- Je zrozumiteľný a preto by mal pomenovávať svoje stavy a prechody slovníkom systému.
- Nie je zanorený príliš hlboko (vnáranie podstavov do jednej, alebo dvoch úrovní ošetrí väčšinu komplexných požiadaviek na chovanie systému).
- Používa simultánne podstavy úsporne. Tieto stavy znižujú zrozumiteľnosť a čitateľnosť stavu. Stavové automaty sú primárne určené k tomu, aby boli na ich základe naprogramované programy. Simultánne podstavy sa ale ťažko vyjadrujú v jazyku programovania a v niektorých programovacích jazykoch toto vôbec nemusí byť možné.

4. XMI: Vymieňanie UML modelov

XMI je skratkou za XML Metadata Interchange a je oficiálnou OMG špecifikáciou pre vymieňanie modelových informácií medzi modelovacími nástrojmi. Verzia 1.0 bola predstavená v júni v roku 2000, verzia 2.0 v roku 2003 a aktuálna verzia 2.4.1 bola vydaná v auguste v roku 2011. Verzie 2.x sa od verzií 1.x radikálne odlišujú. Ako štandard je XMI časťou Meta-Object Facility (MOF) od Object Management Group (OMG) a je tak spojený s inými štandardmi súvisiacimi s MOF, ako UML a Model-driven architecture (MDA).

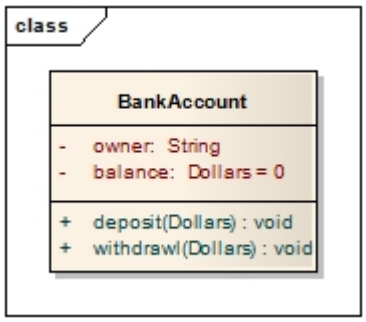
XMI je XML štandard. Ako každý XML štandard, môže byť XMI prehliadaný v normálnom textovom prehliadači a je pre človeka čitateľný. Môže byť čítaný inými modelovacími nástrojmi a uložený ako XML dokument, alebo môžu byť jeho elementy pridané do databázy. Na jeho spracovanie nie sú potrebné žiadne špeciálne nástroje okrem modelovacieho nástroja a prekladača.

Paradoxne problémom štandardu XMI je práve jeho obrovská univerzálnosť a šírka použitia. Tá spôsobuje, že každý z modelovacích nástrojov si daný jazyk interpretuje v rámci možností po svojom a týmpádom sa zo štandardizovaného formátu stáva formát takmer neprenosný. V praxi je veľmi bežná situácia, že XMI súbor vyexportovaný jedným nástrojom druhý nenačíta správne, alebo úplne. Preto pre účely tejto práce bol formát XMI obmedzený na formát exportovaný nástrojom Enterprise Architect (verzie 7.1 až 9.0) a všade ďalej v texte, kde sa hovorí o tomto formáte a jeho špecifikách sa hovorí o informáciách na základe skúmania výstupného XMI súboru z programu Enterprise architect. Je veľmi pravdepodobné, že väčšina funkcionality programu bude kompatibilná aj s väčšinou exportných formátov iných programov.

4.1 Serializované UML

Základnú štruktúru XMI súboru si predvedieme na jednoduchom príklade. Obrázok 4.1 ukazuje triedu pomenovanú "BankAccount", ktorá má dve vlastnosti (`owner : String` a `balance : Dollars = 0`) a dve metódy (`deposit (amount : Dollars)` a `withdrawl (amount : Dollars)`). Ide o diagram triedy zakreslený v UML 2.0.

Jedna možnosť, ako nahliadať na XMI je ako na metódu na serializovanie



Obrázek 4.1: Trieda BankAccount v UML 2.0

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
  <xmi:Documentation exporter="Enterprise Architect" exporterVersion="6.5"/>
```

Obrázek 4.2: Hlavičková sekcia XMI

UML diagramov. Každý UML element môže byť reprezentovaný ako text, uložený a znova otvorený v modelovacom nástroji, rozšírený a opäť uložený ako text. Tento prístup je veľmi podobný procesu serializovania objektu ako SOAP správy.

XMI dokument pozostáva z troch separátnych sekcií. Prvá sekcia je hlavičková sekcia (obrázok 4.2) obsahujúca informácie o verziách štandardov a nástroji, ktorý ju vytvoril.

Ako je uvedené v tagu xmi:XMI, tento model používa XMI verzie 2.1 k reprezentovaniu UML modelu verzie 2.1.

Druhá sekcia reprezentuje UML model (obrázok 4.3). Toto je miesto, v ktorom je uložená logická informácia. Táto sekcia obsahuje informácie, ktoré sa používajú na získanie mien a atribútov rozličných modelovaných položiek.

V kóde na obrázku si je možné všimnúť, že každý element má naprieč celým XML dokumentom unikátne ID, xmi.id.

`Visibility` je logická viditeľnosť UML elementu. Ide tu o viditeľnosť v programe a nie o viditeľnosť daného elementu v rámci diagramu.

Tretia sekcia definuje ako je model zobrazovaný modelovacím nástrojom. Rôzni predajcovia modelovacích nástrojov definujú tieto sekcie rôzne. Nástroj Enterprise Architect používa na popísanie, kde sa UML elementy zobrazujú a v akom štýle, namespace `XMI:Extension` (obrázok 4.4). Zahrnuté sú aj špeciálne informácie špecifické pre Enterprise Architect, ktoré môžu byť exportované do XMI dokumentu.

Kód na obrázku zahŕňa elementy `diagram`, ktorých podelementy a ich atribúty definujú geometrické zostavenie diagramu.

Obrázek 4.3: UML model v XMI

4.2 XMI a stavové automaty

Stavový automat je v XMI reprezentovaný ako element `<packagedElement>` s atribútom `xmi:type` nastaveným na hodnotu `uml:StateMachine`. Potomkom tohto elementu je v XML štruktúre element `region`, ktorý v sebe ohraničuje elementy definujúce jednotlivé stavy stavového automatu a prechody medzi nimi.

Stav stavového automatu je reprezentovaný elementom `subvertex` s atribútom `xmi:type` nastaveným na hodnotu `uml:State`. Pomocou atribútu `name` je definovaný názov daného stavu. Dôležitým je atribút `isSubmachineState`, ktorý ak je nastavený na `true` indikuje, že tento stav je **skladaným stavom** a že máme očakávať podstavy. V takomto prípade je potomkom elementu `subvertex` element `region` a ten môže obsahovať opäť všetky elementy definujúce jednotlivé podstavy tohoto skladaného stavu. Ak má element `subvertex` atribút `isSubmachineState` nastavený na hodnotu `false` ide o jednoduchý stav. V skúmaných exportoch programu **Enterprise Architect** (verzie 7.1 až 9.0) sa ale vyskytla jedna anomália - v prípade, že bol stav označený ako skladaný a teda hodnota `isSubmachineState` bola nastavená na `true` automat ignoroval vytvorené regióny daného stavu a všetky podstavy dával do jedného predvoleného regiónu. Toto znemožňovalo vytváranie simultánnych podstavov. Keď bol však stav nastavený ako jednoduchý (`isSubmachineState` malo hodnotu `false`) a boli v ňom vytvorené regióny, v ktorých boli stavy, boli tieto regióny akceptované a stavy do nich správne pridelené. Z tohto dôvodu bol parser upravený tak, aby si s týmto problémom vedel poradiť a stav, ktorý má definované aspoň dva regióny pokladá za skladaný so simultánnymi podstavmi aj ak má atribút `isSubmachineState`

```
<xmi:Extension extender="Enterprise Architect" extenderID="6.5">
  <elements>
    <element xmi:idref="EAPK_34097722_2C21_484e_9145_B94468D4D552">
      <packageproperties/>
      <paths/>
      <times lastloaddate="2011-09-20 21:19:41"/>
      <flags iscontrolled="FALSE" isprotected="FALSE" usedtd="FALSE" logxml="FALSE" packageFlags="isModel=1;"/>
    </element>
    <element xmi:idref="EAID_2659689F_C3D8_44f9_B4D2_80554E596CAB" xmi:type="uml:Class" name="BankAccount" scope="public">
      <model package="EAPK_764268A7_C9D2_4e93_B3C7_D8F613A90A15" tpos="0" ea_localid="28" ea_eleType="element"/>
      <properties isSpecification="false" sType="Class" nType="0" scope="public" isRoot="false" isLeaf="false" isAbstract="false" isActive="false"/>
      <project author="lubbo" version="1.0" phase="1.0" created="2011-12-02 18:11:29" modified="2011-12-02 18:11:35" complexity="1" status="Proposed"/>
      <code gentype="Java"/>
      <style appearance="BackColor=-1;BorderColor=-1;BorderWidth=-1;FontColor=-1;VSwimLanes=1;HSwimLanes=1;BorderStyle=0;"/>
      <modelDocument/>
      <tags/>
      <xrefs value="$XREFPROP=$XID={8B6A3B6E-5879-47f2-916B-9D9F692B3604}$XID;$NAM=CustomProperties$NAM;$TYP=element property$TYP;$VIS=Public$VIS;$DES=@PROP=@NAME=isActive@ENDNAME;
          @TYPE=Boolean@ENDTYPE;@VALU=@ENDVALU;@PRMT=@ENDPRMT;@ENDPROP;$DES;$CLT={2659689F-C3D8-44f9-B4D2-80554E596CAB}$CLT;$SUP=&lt;none&gt;$SUP;$ENDXREF;"/>
      <extendedProperties tagged="0" package_name="Class Model"/>
      <attributes>
        <attribute xmi:idref="EAID_827A9465_8639_4842_8CAB_D3660411DA58" name="owner" scope="Private">
          <initial/>
          <documentation/>
          <model ea_localid="2" ea_guid="{827A9465-8639-4842-8CAB-D3660411DA58}"/>
          <properties type="String" derived="0" collection="false" duplicates="0" changeability="changeable"/>
          <coords ordered="0"/>
          <containment containment="Not Specified" position="0"/>
          <stereotype/>
          <bounds lower="1" upper="1"/>
          <options/>
          <style/>
          <styleex value="IsLiteral=0;volatile=0;"/>
          <tags/>
          <xrefs/>
        </attribute>
      </attributes>
      <operations>
        <operation xmi:idref="EAID_B8F015AB_BA30_43cc_B04B_CFA6AC8525B7" name="deposit" scope="Public">
          <properties position="0"/>
          <stereotype/>
          <model ea_guid="{B8F015AB-BA30-43cc-B04B-CFA6AC8525B7}" ea_localid="1"/>
          <type type="void" const="false" static="false" isAbstract="false" synchronised="0" concurrency="Sequential" returnarray="0" pure="0" isQuery="false"/>
          <behaviour/>
          <code/>
          <style/>
          <styleex/>
          <documentation/>
          <tags/>
          <parameters>
            <parameter xmi:idref="EAID_RETURNID_BA30_43cc_B04B_CFA6AC8525B7" visibility="public">
              <properties pos="0" type="void" const="false" ea_guid="{RETURNID-BA30-43cc-B04B-CFA6AC8525B7}"/>
              <style/>
              <styleex/>
              <documentation/>
              <tags/>
              <xrefs/>
            </parameter>
          </parameters>
          <xrefs/>
        </operation>
      </operations>
    </element>
  </elements>
  <connectors/>
  <diagrams>
    <diagram xmi:id="EAID_9F12E985_AAD1_4f4a_927E_B9115BD6F2BA">
      <model package="EAPK_764268A7_C9D2_4e93_B3C7_D8F613A90A15" localID="4" owner="EAPK_764268A7_C9D2_4e93_B3C7_D8F613A90A15"/>
      <properties name="Model" type="Logical"/>
      <project author="lubbo" version="1.0" created="2011-09-20 21:20:14" modified="2011-12-02 18:37:04"/>
      <style1 value="ShowPrivate=1;ShowProtected=1;ShowPublic=1;HideRelationships=0;Locked=0;Border=1;HighlightForeign=1;PackageContents=1;SequenceNotes=0;ScalePrintImage=0;
          PPgs.cx=1;PPgs.cy=1;DocSize.cx=827;DocSize.cy=1169;ShowDetails=0;Orientation=P;Zoom=100;ShowTags=0;OpParams=1;VisibleAttributeDetail=0;ShowOpRetType=1;ShowIcons=1;
          CollabNums=0;HideProps=0;ShowReqs=0;ShowCons=0;PaperSize=9;HideParents=0;UseAlias=0;HideAtts=0;HideOps=0;HideStereo=0;HideElemStereo=0;ShowTests=0;ShowMaint=0;
          ConnectorNotation=UML 2.1;ExplicitNavigability=0;AdvancedElementProps=1;AdvancedFeatureProps=1;AdvancedConnectorProps=1;ShowNotes=0;SuppressBrackets=0;
          SuppConnectorLabels=0;PrintPageHeadFoot=0;ShowAsList=0;"/>
      <style2 value="ExcludeRTF=0;DocAll=0;HideQuals=0;AttPkg=1;ShowTests=0;ShowMaint=0;SuppressFOC=1;MatrixActive=0;SwimlanesActive=1;MatrixLineWidth=1;MatrixLocked=0;
          TConnectorNotation=UML 2.1;TExplicitNavigability=0;AdvancedElementProps=1;AdvancedFeatureProps=1;AdvancedConnectorProps=1;ProfileData=;MDGDgm=;STBLDgm=;ShowNotes=0;
          VisibleAttributeDetail=0;ShowOpRetType=1;SuppressBrackets=0;SuppConnectorLabels=0;PrintPageHeadFoot=0;ShowAsList=0;"/>
      <swimlanes value="locked=false;orientation=0;width=0;inbar=false;names=false;color=0;bold=false;fcol=0;;cls=0;"/>
      <matrixitems value="locked=false;matrixactive=false;swimlanesactive=true;width=1;"/>
      <extendedProperties/>
      <elements>
        <element geometry="Left=358;Top=297;Right=492;Bottom=398;" subject="EAID_2659689F_C3D8_44f9_B4D2_80554E596CAB" seqno="1" style="DUID=FC74FB6D;"/>
      </elements>
    </diagram>
  </diagrams>
</xmi:Extension>
</xmi:XMI>
```


Obrázek 4.4: Blok XMI:Extension

nastavený na `false`.

Okrem spomínaného elementu `region`, ktorý slúži na definovanie podstavov, môže element `subvertex` obsahovať a typicky aj obsahuje jeden, alebo niekoľko elementov `incoming` a jeden, alebo niekoľko elementov `outgoing`. Tieto elementy označujú, že v danom stave začína (element `outgoing`), alebo končí (element `incoming`) nejaký prechod. Tieto elementy majú len jeden atribút - `xmi:idref`, ktorého hodnota slúži ako odkaz na element `transition` s rovnakou hodnotou atribútu `xmi:id`.

Element `transition` bližšie popisuje daný **prechod**. Dôležité atribúty sú atribúty `source` a `target`, ktoré oba obsahujú hodnoty odkazujúce na element `subvertex` s rovnakou hodnotou atribútu `xmi:id`. Takto sú označené zdrojový (atribút `source`) a cieľový (atribút `target`) stav medzi ktorými je daný prechod definovaný.

Element `transition` môže mať ďalej potomka element `trigger`, ktorý pomocou hodnoty atrubútu `xmi:idref` odkazuje na element `packagedElement` s atribútom `xmi:type` nastaveným na hodnotu `uml:Trigger`. Tento element slúži na špecifikáciu aktivátoru daného prechodu. Bližšie si ho popíšeme nižšie v texte.

Okrem elementu `trigger` môže element `transition` obsahovať element `guard` ktorého potomkom je element `specification`, ktorý v atribúte `body` obsahuje špecifikáciu **kontrolnej podmienky**, pri ktorej môže k prechodu dôjsť.

Pre úplnosť ešte musíme uviesť ako sa v rámci elementu `region` zadefinujú štartovací a koncový stav.

Štartovací stav je reprezentovaný elementom `subvertex` s atribútom `xmi:type` nastaveným na hodnotu `uml:Pseudostate` a atribútom `kind` nastaveným na hodnotu `initial`.

Koncový stav je reprezentovaný elementom `subvertex` s `xmi:type` nastaveným na hodnotu `uml:FinalState`.

Ako už bolo spomenuté v texte vyššie, okrem elementu `packagedElement` s atribútom `xmi:type` nastaveným na hodnotu `uml:StateMachine`, sa v XMI súbore nachádzajú ešte ďalšie elementy `packagedElement`, ktorých atribút `xmi:type` je nastavený na hodnotu `uml:Trigger`. Tieto elementy slúžia na definíciu **aktivátorov udalostí**.

Každý takýto element má ako potomka element `event`, ktorý definuje udalosť, pri ktorej sa prechod aktivuje. Element `event` môže byť niekoľkých typov, ktoré sú definované jeho atribútom `xmi:type`. Podľa toho, o akú udalosť ide, môže tento atribút nadobúdať hodnoty:

- `uml:CallEvent` pre udalosť volania

- `uml:TimeEvent` pre udalosť vyvolanú chodom času. Podelementom pre tento typ udalosti je element `when`, ktorý v atribúte `expr` obsahuje definíciu časovej podmienky spúšťajúcej udalosť.
- `uml:SignalEvent` pre signálnu udalosť
- `uml:ChangeEvent` pre udalosť zmeny stavu. Podelementom pre tento typ udalosti je element `changeExpression`, ktorý v atribúte `symbol` obsahuje definíciu logickej podmienky spúšťajúcej udalosť.

5. Riešenie problému

Jednotlivé aspekty riešenia vychádzajú zo zadaného cieľa práce a z požiadaviek zadania. Popíšeme teraz voľby ktoré boli vykonané a riešenie definujú.

5.1 Parser

Automat je programu odovzdávaný prostredníctvom vstupného súboru, v ktorom je automat zapísaný v určitom formáte. Požiadavkou zadania bolo, aby program nebol obmedzený na jeden typ vstupného súboru, ale aby bolo možné si implementovať vlastný parser vstupného súboru. Aby bolo toto možné, musí byť automat najskôr zo vstupného súboru prevedený na interný jednotný formát. Za týmto účelom bol vytvorený objektový model automatu a automat je najskôr prevedený zo vstupného súboru na tento model. Toto je práve úloha parsera. Implementovať nový parser znamená implementovať dané povinné rozhranie s metódou, ktorá na vstupe dostane vstupný súbor a na výstupe vráti objektový model patričného automatu. Na identifikáciu všetkých dostupných parserov sa použije reflexia.

5.2 Generátor

Na generátor kódu boli kladené hneď dve dôležité požiadavky - aby umožňoval rozšírenie o moduly pre ďalšie výstupné jazyky a aby programátor u týchto modulov nemusel programovať logiku generovania kódu zo stavového automatu.

Riešenie prvej požiadavky nie je principiálne tak zložité - stačí definovať presné rozhranie, ktoré tieto moduly musia implementovať a potom v aplikácii pomocou reflexie tieto moduly detekovať a ponúknuť užívateľovi zvolenie použitia ľubovoľného z nich.

Druhá požiadavka už celkový návrh aplikácie ovplyvňuje výraznejšie. Aby nemusel programátor modulu implementovať logiku generovania kódu, je potrebné, aby generátor vedel generovať kód bez ohľadu na to, do akého jazyka ho generuje a potom už použil len daný modul na vygenerovanie konkrétneho výstupného jazyka.

Z tohto dôvodu bol vytvorený objektový model programu a generátor v prvej fáze generuje kód z modelu automatu do modelu programu. Programátor, ktorý potom programuje modul na generovanie výstupu do príslušného jazyka potom už len implementuje rozhranie, ktoré poskytuje metódy na vyjadrenie jednotlivých javov v danom programovacom jazyku. Tieto javy sú obmedzené

na úroveň výrazov typu "vypíš kód podmienky, ktorá overuje rovnosť dvoch zadaných prvkov", "vypíš kód príkazu if so zadanou podmienkou" a podobne. Objektový model programu je potom pomocou volaní týchto metód transformovaný do daného konkrétneho jazyka.

5.3 Prevod automatu na kód

Najdôležitejším rozhodnutím celej práce je rozhodnutie ako sa jednotlivé javy stavových automatov budú prevádzať na kód. Pre celkovú komplexnosť jazyka UML a a stavových automatov v ňom musela byť množina týchto javov čiastočne zredukovaná, pretože niektoré z nich by si vyžadovali nedeterministické chovanie výstupného programu, alebo asynchrónny chod programu vo viacerých vláknach. Keby mali byť tieto javy zachované, bola by výrazne obmedzená množina možných výstupných jazykov, čo by kolidovalo s požiadavkou umožniť programátorom rozširovať program o svoje vlastné výstupné moduly.

Boli vykonané nasledujúce obmedzenia:

1. Automaty nepodporujú podstavy s históriou.
2. Prechody medzi stavmi môžu byť len s aktivátorom prechodu typu `signál`, alebo bez aktivátora.
3. Každý zložený podstav musí obsahovať štartovací stav.
4. Stavy nebudú obsahovať aktivity a asynchrónne udalosti.
5. Stavy nebudú obsahovať interné prechody.

V prípade, že sa vo vstupnom automate bude nachádzať podstav bez štartovacieho stavu, bude užívateľovi oznámená chybová hláška a vstupný automat nebude spracovaný. V prípade výskytu ostatných javov, budú tieto na vstupe ignorované a kód bude vygenerovaný tak, ako keby sa tieto na vstupe nenachádzali. Poslednou požiadavkou na vstup je, že každý automat bude patriť pod nejakú triedu a že každá takáto trieda bude obsahovať práve jeden automat.

Výstupom by mal byť kód, ktorý bude umožňovať simuláciu práce automatu, tj. používateľ automatu mu bude môcť zasielať udalosti a automat na základe toho bude meniť svoje stavy. Pri prevode z objektového modelu automatu na objektový model programu sa uplatňujú nasledovné pravidlá:

1. Pre každú triedu z modelu automatu bude vytvorená jedna trieda v modeli programu. Do tejto triedy budú prebrané aj vlastnosti definované v modeli automatu, pretože je možné ich použitie v akciách automatu.

2. Bude vytvorený výčtový typ (enum), ktorý bude pre každý stav automatu (vrátane štartovacích a koncových stavov) obsahovať jednu hodnotu. Jednotlivé hodnoty sú pomenovávané tak, ako stavy automatu. U štartovacích a koncových stavov by to ale mohlo spôsobiť duplicity, pretože tieto bývajú obvykle pomenované Initial a Final a to aj opakovane v rôznych podstavoch automatu. Ak toto nastane, ich názvy sú rozšírené o suffix, ktorý zabezpečí unikátnosť hodnoty výčtového typu.

3. Bude vytvorený výčtový typ (enum), ktorý bude pre každú udalosť vyvolávajúcu prechod obsahovať jednu hodnotu. Jednotlivé hodnoty sú pomenovávané tak, ako príslušné udalosti.

4. Pre stavový automat a každý z jeho podautomatov bude trieda obsahovať členskú premennú typu list položiek z výčtového typu stavov. Pre hlavný automat bude táto premenná pomenovaná `currentState`, pre podautomaty bude tento názov doplnený o suffix s názvom nadradeného stavu, ktorému daný podautomat patrí, prípadne v prípade paralelných podstavov ešte o názov regiónu v ktorom sa tento podautomat nachádza.

5. Na začiatku práce s automatom sa tieto listy zinicializujú a do každého sa vloží hodnota odpovedajúca štartovaciemu stavu príslušného automatu.

6. Pre každú prechodovú akciu medzi dvoma stavmi bude vytvorená metóda obsahujúca jej kód.

7. Pre každú vstupnú akciu stavu bude vytvorená metóda obsahujúca jej kód.

8. Pre každú výstupnú akciu stavu bude vytvorená metóda obsahujúca jej kód.

9. Bude vytvorená metóda `processEvent` ktorá na vstupe dostane hodnotu z číselníka udalostí a na základe tejto udalosti vykoná prechod nad automatom. Návratový typ tejto metódy bude boolean a vrátená hodnota bude signalizovať, či sa danú udalosť podarilo obslúžiť.

Spomínaná metóda `processEvent` je teda metóda, ktorá slúži na simuláciu chodu automatu. Predstava je taká, že užívateľ kódu metódu opakovane

volá v svojom programe a na základe udalostí, ktoré jej zasiela automat priebežne mení svoj stav. Pri skúmaní stavových automatov v jazyku UML boli vykonané nasledovné kľúčové pozorovania:

1. Stavový automat je možné reprezentovať ako switch cez jednotlivé stavy, kde sa postupne hľadá prechod na obslúženie danej udalosti.

2. Keď nejaký prechod opúšťa zložený stav opúšťa aj aktuálne aktívne stavy jeho podautomatov.

3. Prechod môže viesť naprieč viacerými úrovňami jedného automatu aj naprieč viacerými automatmi. Vtedy treba riadne opustiť každý z opúšťaných stavov (tj. zavolať ich výstupné akcie, zrušiť aktívnosť). Takisto treba riadne vstúpiť do každého z novonavštevovaných stavov (tj. zavolať ich vstupné akcie, nastaviť ich ako aktívne v danom nadradenom automate).

4. Dôležité je preskúmať chovanie simultánnych podautomatov, pretože tu môžu byť určité špecifiká a môže tu nastať najviac rozdielnych situácií, ktoré musí vedieť automat obslúžiť. Ako uvádzajú Arlow a Neustadt (1) paralelný podstav je možné opustiť troma rôznymi spôsobmi:

   (a) Všetky podautomaty skončia svoje úlohy - to je implicitné spojenie.

   (b) Jeden z podautomatov prejde do stavu *mimo* kontext nadstavu. To *nespôsobí* spojenie - v tomto prípade nedôjde k synchronizácii podautomatov.

   (c) Keď má nejaký nadradený stav tiež výstupný prechod, ktorý je dedený oboma podstavmi. Vďaka tomu je možné stav opustiť okamžite.

   Z 4.(a) vyplýva, že ak má byť stav opustený normálnym prechodom bez udalosti, musí sa počkať kým sa jeho jednotlivé podautomaty zosynchronizujú a všetky vstúpia do svojich finálnych stavov. Z 4.(b) zas vyplýva, že pri nesynchronizovanom opustení zloženého stavu sa kontext automatu rozdelí na dve vetvy - aktívny bude stav do ktorého smeroval prechod opúšťajúci jeden z podautomatov a zároveň budú aktívne aj stavy v ďalších podautomatoch zloženého stavu. Aktívnymi na tej istej úrovni alebo na rôznych úrovniach sa môžu teda stať viaceré stavy súčasne. Preto vzhľadom k tomu, že cieľom práce je podporovať aj paralelné podstavy, nebude stačiť na reprezentovanie aktuálneho stavu daného automatu jedna jednohodnotová premenná, ale bude to musieť

byť zložená premenná, ktorá dokáže súčasne udržiavať viacero hodnôt. Z testov na rovnosť stavu sa budú musieť stať testy na náležanie daného stavu do množiny aktuálne aktívnych stavov. Preto ani nie je možné použiť na reprezentovanie automatu príkaz switch, ako tomu je u jednoduchších generátorov spomínaných v kapitole 2, ale bude sa musieť použiť séria if blokov, ktorá tento pomyselný príkaz switch bude simulovať.

Vzhľadom k zisteným pozorovaniam bol na obsluhu udalosti automatom navrhnutý nasledujúci postup, ktorý udržiava automat v konzistentnom stave a pokrýva zistené požiadavky pozorovaní:

1. Na začiatku metódy sa zadeklaruje booleanová premenná `eventConsumed` a jej hodnota sa nastaví na `false`.
2. Pre každý stav sa vygeneruje blok `if (currentState.Contains(state))`, kde `state` je hodnota z výčtového typu stavov.
3. V každom tomto bloku sa vykonajú nasledovné akcie:
   (a) Ak ide o zložený stav, tak pre každý z jeho podautomatov sa vygeneruje kód rovnako ako v bodoch 2. až 3. tohto výčtu, len namiesto listu `currentState` sa požije príslušný list patriaci k danému automatu.
   (b) Pre každú udalosť `event` takú, že pomocou udalosti `event` sa dá vystúpiť z daného stavu, sa vygeneruje blok `if (e == event)` kde `e` je parameter metódy `processEvent`, v ktorom metóda dostane aktuálnu udalosť na spracovanie.
   (c) V každom tomto bloku sa vykonajú nasledovné akcie:
      i. Premenná `eventConsumed` sa nastaví na `true`.
      ii. Ak daný prechod obsahuje nejakú prechodovú akciu, bude zavolaná metóda, ktorá volá túto akciu.
      iii. Ak je prechodom opúšťaný zložený stav, je pre všetky jeho aktuálne aktívne podstavy (rekurzívne, do hĺbky) zavolaná metóda volajúca výstupnú akciu daného podstavu, ak má daný podstav definovanú výstupnú akciu.
      iv. Pre každý stav, ktorý je opúšťaný na ceste zo zdrojového do cieľového stavu podľa prechodu vyvolaného udalosťou `event` je zavolaná metóda volajúca výstupnú akciu stavu, ak tento stav nejakú výstupnú akciu má zadeklarovanú. Pojem "opúšťaný na ceste zo zdrojového do cieľového stavu" je tu použitý preto,

lebo prechod môže prechádzať cez jednotlivé úrovne stavového automatu a tak sa môže kontext vynoriť jedným prechodom nielen zo zdrojového stavu, ale aj z jeho nadstavov.

v. Ak je prechodom opúšťaný zložený stav, je pre všetky jeho aktálne aktívne podstavy (rekurzívne, do hĺbky) daný podstav odstránený z príslušného `currentState` listu príslušného nadradeného automatu a do listu je pridaný štartovací stav daného nadradeného automatu.

vi. Pre každý stav, ktorý je opúšťaný na ceste zo zdrojového do cieľového stavu podľa prechodu vyvolaného udalosťou `event` je daný stav odstránený z príslušného `currentState` listu príslušného nadradeného automatu a do listu je pridaný štartovací stav daného nadradeného automatu.

vii. Pre každý stav, do ktorého sa vstupuje na ceste zo zdrojového do cieľového stavu podľa prechodu vyvolaného udalosťou `event` je z príslušného `currentState` listu príslušného nadradeného automatu odobraný štartovací stav. Pojem "do ktorého sa vstupuje na ceste zo zdrojového do cieľového stavu" je tu použitý preto, lebo prechod môže prechádzať cez jednotlivé úrovne stavového automatu a tak sa môže kontext zanoriť jedným prechodom aj do iných podautomatov, ako je automat v ktorom sa nachádza zdrojový stav.

viii. Pre každý stav, do ktorého sa vstupuje na ceste zo zdrojového do cieľového stavu podľa prechodu vyvolaného udalosťou `event` je do príslušného `currentState` listu príslušného nadradeného automatu pridaný tento stav.

ix. Pre každý stav, do ktorého sa vstupuje na ceste zo zdrojového do cieľového stavu podľa prechodu vyvolaného udalosťou `event` je zavolaná metóda volajúca vstupnú akciu stavu, ak tento stav nejakú vstupnú akciu má zadeklarovanú.

x. Udalosť je týmto obslúžená, metóda `processEvent` vráti hodnotu `true`

(d) Ak existuje prechod bez udalosti vedúci z daného stavu `state` vygeneruje sa príkaz `if` s podmienkou testujúcou či premenná `eventConsumed` má hodnotu `false` (to znamená že žiaden prechod z daného aktívneho stavu zatiaľ neobslúžil udalosť), použije tento prechod spôsobom zhodným s postupom v bode (c) tohto výčtu. V prípade, že stav `state` je skladaný stav, bude táto podmienka doplnená o overenie, či všetky priame podautomaty daného automatu sú vo svojom koncovom stave. Z každého stavu by mal viesť

maximálne jeden takýto prechod, ak ich existuje viac, ako jeden, je to chyba dizajnu. Generátor ich pretransformuje na kód všetky, ale takéto nadbytočné vetvy nebudú mať vo výstupnom kóde zmysel - nikdy nebudú môcť byť aktivované.

4. Ak doteraz udalosť nebola obslúžená, metóda `processEvent` vráti `false`.

6. Architektúra a implementácia riešenia

Z popisu generátora tak, ako je podaný v predošlej kapitole vyplývajú pre výsledný program určité obmedzenia. Napríklad vstupný automat by musel byť z objektového modelu automatu prevádzaný na konkrétny objektový model programu. Tento model by musel byť jednoznačný a preto by bol program zrejme modelovaný pomocou štandardných prostriedkov objektovo orientovaného programovania ako sú triedy, metódy, výčtové typy a podobne. Na tento model by následne bola úzko naviazaná druhá fáza generovania kódu, tj. preklad do konkrétneho výstupného jazyka. Keďže táto fáza by už pracovala s konkrétnym typom modelu, obmedzovala by ďalej možnosti výstupného jazyka na určitú typovú množinu. Toto riešenie je teda obmedzujúce, ale zároveň umožňuje splnenie požiadavky, aby programátor, ktorý chce vytvoriť nový modul pre nový výstupný jazyk nemusel ísť do hĺbky generovania kódu, ale definoval len pravidlá zápisu kódu v danom jazyku. Toto je dodržané, ale daný jazyk musí spĺňať určité charakteristiky - musia to byť jazyky, ktoré poznajú pojmy ako trieda, metóda, výčtový typ a ktoré sú orientované tak, že metódy, výčtové typy a vlastnosti sa deklarujú v triede. Aby toto obmedzenie nebolo v konečnom dôsledku definitívne, bol pri návrhu architektúry program navrhnutý tak, aby skúsenejšiemu programátorovi umožnil ísť pri implementácii modulu hlbšie a tým dosiahol širšie možnosti generovania za cenu potreby preimplementovania ďalších súčastí kódu. Program z hľadiska architektúry pozostáva z niekoľkých komponent (balíčkov).

6.1 Model programu (CodeModel)

Na základe postupu uvedeného v 5.3 vyplývajú na výstupný programovací jazyk určité kritériá a predpoklady, ktoré by mal spĺňať. Tieto kritériá sú pomerne obecné a mnoho bežných jazykov sa im dokáže prispôsobiť. Zároveň tieto kritériá stanovujú, čo všetko treba vedieť v modeli programu namodelovať, aby sa prostredníctvom neho dal namodelovať výstupný program. Zistené boli nasledovné požiadavky:

1. Musí obsahovať triedy.
2. Musí byť možné zadefinovať výčtový typ, alebo jeho ekvivalent.
3. Musí byť možné zadefinovať vlastnosť triedy a to jednak nejakého jednoduchého typu (integer, float, boolean ...), tak aj výčtového typu (pre

zadeklarovaný výčtový typ) a typu list položiek daného typu.

4. Musí byť možné zadefinovať metódu aj so vstupnými parametrami a návratovým typom.

5. Musí byť možné zadefinovať v metóde lokálnu premennú ľubovoľného typu.

6. Musí byť možné s vlastnosťou typu list vykonávať nasledovné operácie:

   (a) inicializácia

   (b) pridanie hodnoty do listu - musí byť možné pridať tú istú hodnotu do listu aj opakovane a táto hodnota sa musí v danom liste nachádzať daný počet krát.

   (c) odstránenie jedného výskytu danej hodnoty z listu

   (d) odstránenie všetkých hodnôt z listu

   (e) test či list obsahuje danú hodnotu

   (f) spočítať počet všetkých výskytov danej hodnoty v liste

   (g) po úvahe ešte bola pridaná operácia - vymazanie všetkých výskytov danej hodnoty z listu aj keď táto operácia v popisovanom algoritme použitá nie je, ale užívateľ by mohol vo svojich úpravách mať záujem ju použiť

7. Musí byť možné priradiť hodnotu do lokálnej premennej metódy alebo do vlastnosti triedy. Pod hodnotou sa chápu jednak literály (true, 0, null ...) ale jednak aj hodnoty iných vlastností a premenných. Okrem toho z účelového hľadiska musí byť možné priradiť aj hodnotu výsledku operácie na spočítanie počtu všetkých výskytov danej hodnote v liste, táto operácia by totiž inak nemala zmysel.

8. Musí byť možné zapísať príkaz if aj s prípadnou else vetvou. Ako podmienku tohto príkazu musí byť možné použiť test na rovnosť, test či je jedna hodnota väčšia ako druhá, test či platia dve podmienky súčasne a test či list obsahuje danú hodnotu.

9. Musí byť možné vrátiť z metódy hodnotu a to jednak v podobe literálu, ako aj v podobe lokálnej premennej metódy alebo vlastnosti triedy.

10. Musí byť možné vykonať príkazy v cykle s pevným počtom opakovaní (for cyklus).

11. Musí byť možné zavolať metódu a zadefinovať jej ohodnotenia jednotlivých parametrov. Ako ohodnotenie opäť môže poslúžiť literál, lokálna premenná metódy, alebo vlastnosť triedy.

12. Po úvahe bol ešte pridaný príkaz switch pre prípad ak by ho programátor nového modulu chcel využiť. Tento príkaz by bol napríklad veľmi užitočný ak by sa programátor rozhodol nepodporovať simultánne podstavy. Potom by bolo možné celý automat reprezentovať ako jeden veľký switch príkaz s prípadnými podswitchmi pre podautomaty.

Z ohľadom na tieto požiadavky boli navrhované triedy a rozhrania balíčka `CodeModel`, ktorý obsahuje triedy a rozhrania slúžiace na reprezentáciu objektového modelu programu. Centrálnym rozhraním je rozhranie `ICodeModel`, ktoré je v tomto balíčku implementované jednou triedou `Class`. Preto ako sa uvádza vyššie je štandardný objektový model programu orientovaný okolo triedy. Trieda `Class` má svoj definovaný názov a obsahuje list objektov implementujúcich rozhranie `IProperty` (vlastnosti triedy), list objektov triedy `Method` (metódy triedy) a list objektov implementujúcich rozhranie `IEnumType` (výčtové typy triedy).

Všetko do čoho môže byť v rámci programu priradená hodnota musí implementovať rozhranie `IAssignable`. Toto rozhranie implementuje v modeli abstraktná trieda `BasicValueHolder` ktorá má definovaný typ (implementáciu rozhrania `IType`) a názov. Potomkami tejto triedy sú triedy `Parameter`, `Property` a `Variable`. Trieda `Property` rozširuje možnosti o možnosť definovať vlastnosť triedy ako predinicalizovanú (tzn. že rovno s deklaráciou je jej nastavená hodnota) pomocou vlastností `bool Preassigned` a `IEvaluable PreassignValue`.

Metódy sa skladajú z príkazov a tak trieda `Method` obsahuje okrem názvu, návratového typu (implementácia rozhrania `IType`) a listu parametrov (trieda `Parameter`) aj list objektov implementujúcich rozhranie `ICommand`.

V modeli ho implementuje `Assignment` priradenie (vlastnosti `IEvaluable Source` a `IAssignable Destination`), trieda `DeclareVariable` reprezentujúca deklaráciu premennej (vlastnosť `Variable VariableDeclared`), `IfElse` reprezentujúca podmienený príkaz if - else (vlastnosti `ICondition Condition`, `LinkedList<ICommand> IfBody` a `LinkedList<ICommand> ElseBody`), trieda `ListAddCall` reprezentujúca volanie pridania hodnoty do listu (vlastnosti `IList List` a `IEvaluable Value`), trieda `ListClearCall` reprezentujúca volanie vyprázdnenia listu (vlastnosť `IList List`), trieda `ListInitCall` reprezentujúca volanie inicializácie listu (vlastnosť `IList List`), `ListRemoveCall` reprezentujúca volanie odstránenia hodnoty z listu (vlastnosti `IList List` a `IEvaluable Value`), trieda `MethodCall`, ktorá reprezentuje volanie metódy

(vlastnosti `Method MethodCalled` a `Dictionary<Parameter, IEvaluable> Parameters`), `Switch` reprezentujúca príkaz switch (vlastnosti `IEvaluable Value` a `List<Case> Cases`), trieda `ReturnValue` reprezentujúca návrat hodnoty z metódy (vlastnosť `IEvaluable Value`) a trieda `UnresolvedCommand` (vlastnosť `string Text`). Posledná menovaná sa používa na príkazy, ktoré boli definované užívateľom vo vstupnom automate a teda nie je u nich možné rozlíšiť o aké príkazy konkrétne ide. Preto sú použité priamo v textovej forme, ako ich užívateľ zadal. Je teda na užívateľovi, aby zadával príkazy tak, aby boli daným výstupným jazykom spracovateľné, inak automat nebude priamo použiteľný a užívateľ bude musieť výsledný kód doupraviť. Ako je vidno, niektoré príkazy v sebe obsahujú ďalšie príkazy a týmto spôsobom je možné vymodelovať celé bloky kódu.

V podmienených príkazoch triedy `IfElse` je potrebné zadávať podmienku vetvy if, ktorá je reprezentovaná rozhraním `ICondition`.

Implementáciami tohto rozhrania sú `AndCondition` (vlastnosti `ICondition Left` a `ICondition Right`), trieda `EqCondition` (vlastnosti `IEvaluable Left` a `IEvaluable Right`), trieda `ListContainsCondition` (vlastnosti `IList List` a `IEvaluable Value`) a `UnresolvedCondition` (vlastnosť `string Text`). Posledná spomínaná je opäť používaná pre podmienky zadané užívateľom vo vstupnom automate, konkrétne podmienky podmienených prechodov.

Implementáciou rozhrania `IEnumType` je trieda `EnumType`, ktorá okrem názvu daného výčtového typu obsahuje zoznam hodnôt (`IEnumValue`, ktoré môže výčtový typ nadobúdať.

Spomínané rozhranie `IEnumValue` má v modeli implementáciu `EnumValue`.

Všetko čo môže mať hodnotu a teda môže byť vyhodnotiteľné a priraditeľné musí implementovať rozhranie `IEvaluable`. Okrem spomínanej triedy `BasicValueHolder` a jej potomkov ho implementuje rozhranie `IEnumValue` a trieda `SimpleValue` (vlastnosť `object Value`) ktorá reprezentuje jednoduchú hodnotu - literál.

Rozhranie `IList` reprezentuje všetko čo z pohľadu programu môže byť listom a teda na čom sa dajú volať listové operácie. Implementáciou tohto rozhrania je trieda `ListProperty`, ktorá má okrem názvu definovaný ešte typ (trieda `ListType`).

Rozhranie `IProperty` vyčleňuje všetko, čo môže byť vlastnosťou triedy. Implementujú ho spomínané triedy `Property` a `ListProperty`.

Všetko čo musí mať definovaný typ má ako vlastnosť implementáciu rozhrania `IType`. Jeho implementáciami sú trieda `EnumType`, trieda `ListType` a trieda `SimpleType`. Posledná menovaná reprezentuje jednoduchý jednohodnotový typ, ktorý je definovaný vlastnosťou `DefiningType` typu `System.Type` (štandardný typ v C#).

Dôležité je, že ako styčný bod s generátorom je používané práve rozhranie `ICodeModel` a tak si môže programátor v prípade potreby vytvoriť jeho vlastnú implementáciu, ktorá bude modelovať inak štrukturovaný program. Štruktúra tohto balíčka je veľmi rozsiahla pretože musí síce v zjednodušenej ale dostačujúcej forme modelovať program s väčšinou jeho štandardných vlastností. Preto je výpis vyššie skôr len orientačný a podrobnejšie informácie je možné získať preštudovaním generovanej programátorskej dokumentácie alebo uml diagramov v prílohe C.

6.2 Generátor kódu (generator)

Balíček `generator` obsahuje 3 kľúčové rozhrania a štandardné implementácie dvoch z nich.

Rozhranie `ICodeGenerator` je rozhraním na generovanie kódu z objektového modelu programu. Pomocou metódy `SetTranslator` mu je nastavený objekt implementujúci rozhranie `ILanguageTranslator` a s jeho použitím by mal byť generovaný kód z objektu implementujúceho rozhranie `ICodeModel` volaním metódy `GenerateCode`, ktorá vráti textový reťazec obsahujúci príslušný kód.

Rozhranie `ICodeModelBuilder` predpisuje jedinú metódu `GenerateModel`, ktorá na vstupe dostane objektový model triedy obsahujúcej automat (z balíčka `StateMachineModel`) a vráti objektový model programu odpovedajúceho tejto triede v podobe objektu implementujúceho rozhranie `ICodeModel`.

Rozhranie `ILanguageTranslator` poskytuje metódy na vyjadrenie potrebných javov v reči programovacieho jazyka.
Nachádzajú sa tu metódy `PrintEqCondition(string left, string right)`, `PrintAndCondition(string left, string right)` a ďalšie.

Metóda `FormatCode(string code)` zas môže slúžiť k formátovacím úpravám vygenerovaného kódu. Toto rozhranie je styčným bodom medzi programátorom modulu a hlavným programom, pretože toto jediné je skutočne nutné implementovať vždy pre každý nový jazyk. Preto toto rozhranie požaduje ešte dve špecifické metódy `GetHumanReadableName`, ktorá vráti názov daného programátorského jazyka a `GetFileExtension`, ktorá vráti štandardnú príponu zdrojových súborov daného jazyka.

Trieda `DefaultCodeModelBuilder` je štandardnou implementáciou rozhrania `ICodeModelBuilder`. Generuje objektový model programu v podobe objektu triedy `Class` z balíčka `CodeModel`.

Trieda `DefaultGenerator` je implementáciou rozhrania `ICodeGenerator`. Používa objekt implementujúci rozhranie `ILanguageTranslator` na generovanie kódu v štandardnom objektovo orientovanom programovacom jazyku

typu C# či Java. Ako vstupný model očakáva objekt triedy `Class` z balíčka `CodeModel`.

6.3 Parser vstupných súborov (parser)

Ústredným rozhraním balíčka `parser` je rozhranie `IParser`. Jeho metóda `Parse` dostane na vstupe názov vstupného súboru a na výstupe vracia list objektov triedy `Class` z balíčka `StateMachineModel`. Metóda `IdentifyFile` zas dostane názov súboru a jej úlohou je overiť, či je tento súbor príslušným parserom parsovateľný. V prípade problému vráti chybovú hlášku.

6.4 Model stavového automatu (StateMachineModel)

Balíček `StateMachineModel` obsahuje triedy a rozhrania slúžiace na reprezentáciu objektového modelu stavového automatu. Ako už bolo uvedené, každý automat sa viaže k nejakej triede, takže držiteľom automatu je trieda `Class`. Každá `Class` môže mať jednu inštanciu triedy `StateMachine` ktorá modeluje stavový automat. Táto trieda je potomkom triedy `BasicStateMachine`, ktorej druhým potomkom je trieda `SubStateMachine`, ktorá slúži k modelovaniu podautomatu nejakého zloženého stavu. Rozdiel medzi `SubStateMachine` a `StateMachine` je v tom, že `SubStateMachine` má definovaný definovaný svoj rodičovský stav - implementáciu rozhrania `ICompositeState`. Toto rozhranie implementujú dve triedy - `SequentialCompositeState`, ktorá modeluje jednoduchý zložený stav a teda vlastní len jeden podautomat a `ConcurrentCompositeState`, ktorá modeluje paralelný zložený stav a môže obsahovať podautomatov viac. Teoreticky sa dal použiť aj jeden spoločný typ zloženého stavu, kde by bol jednoduchý stav bol len špeciálnou variantou paralelného, pre jednoduchšiu manipuláciu v kóde ale toto bolo rozdelené a tieto dva typy stavov sa rozlišujú. Oba typy zložených stavov dedia po triede `BasicState`, ktorá definuje všetky vlastnosti, ktoré má každý regulérny stav stavového automatu. Jej posledným potomkom je trieda `SimpleState` ktorá reprezentuje obyčajný stav bez podstavov. Špeciálnymi "stavmi", ktoré ale nededia po `BasicState` sú triedy `Start` a `End`. Tieto pseudostavy sú tak približne na rozhraní medzi stavom a niečím iným a tak ich so stavmi spája spoločné rozhranie `IState`, ktoré im umožňuje figurovať v zozname všetkých stavov automatu ale nemôžu napríklad mať vstupné a výstupné akcie. Prechody medzi jednotlivými stavmi modeluje trieda `Transition`.

6.5 Hlavná aplikácia

Hlavná aplikácia je jadro, ktoré poskytuje užívateľovi užívateľské rozhranie na prácu s programom. Toto rozhranie je zapísané prostredníctvom jazyka `XAML` vo `Windows Presentation Foundation`.

Jadro aplikácie po spustení prejde dll súbory podadresára `output_types` a hľadá v nich implementácie rozhraní `ILanguageTranslator`, `ICodeGenerator` a `ICodeModelBuilder`. Implementácia rozhrania `ILanguageTranslator` je povinná a ak sa nenájde tá, tak je obsah daného súboru ignorovaný.

Implementácia rozhrania `ICodeGenerator` a rozhrania `ICodeModelBuilder` povinné nie sú. Ak ich programátor v knižnici dodal, tak sú použité jeho verzie. Ak niektoré z týchto dvoch rozhraní implementované v knižnici nie je, použijú sa štandardné `DefaultGenerator` resp. `DefaultCodeModelBuilder`. Takéto usporiadanie ponúka programátorovi unikátnu možnosť ísť tak do hĺbky, ako je to pre neho potrebné. Ak mu vyhovuje štandardný objektovo orientovaný generátor, stačí aby implementoval translátor čo je veľmi jednoduché. Ak mu vyhovuje objektovo orientovaný model programu vytváraný štandardným `DefaultCodeModelBuilder` ale potrebuje iný generátor, môže implementovať rozhranie `ICodeGenerator` a definovať ako sa má z modelu pomocou metód translátora vygenerovať kód. Ak mu nevyhovuje ani model programu, alebo spôsob, ako sa automat prevádza na tento model, môže implementovať aj rozhranie `ICodeModelBuilder`.

Z takto získaných implementačných modulov potom hlavná aplikácia naplní ponuku na výber výstupného jazyka. K označeniu jazyka slúži volanie metódy `GetHumanReadableName` predpísanej rozhraním `ILanguageTranslator`. Program jednotlivé implementácie rozhraní inštanciuje a pri voľbe výstupného jazyka použije príslušné inštancie, prípadne doplnené o inštancie štandardných implementácií.

Celý návrh je koncipovaný tak, aby translator vedel o výslednom kóde čo najmenej a celá logika prevodu z modelu programu na kód je kladená na generator. Je na ňom, aby volal metódy translatora v takom poradí aby výsledok dával zmysel. Translator napríklad ani nevie zadeklarovať celý kód metódy aj s telom, alebo vnútro príkazov if a for. Poskytuje len rozhranie na vypísanie začiatku a konca metódy či príkazu. S použitím ďalších metód translatora vygeneruje generator telo - sled príkazov a potom ho vloží medzi spomínaný začiatok a koniec. Z tohto dôvodu je žiadané, aby translator aj čo najmenej pracoval s modelom programu, pretože aj ten je teoreticky možné vymeniť. Preto bola jeho znalosť modelu ako takého obmedzená na minimum a kde nejaká spolupráca bola nutná, používajú sa rozhrania, takže vnútornosti modelu bude stále možné si preimplementovať. Väčšina metód translatora, ale pracuje s čistými textovými reťazcami. Ako textový reťazec sú zadávané aj ty-

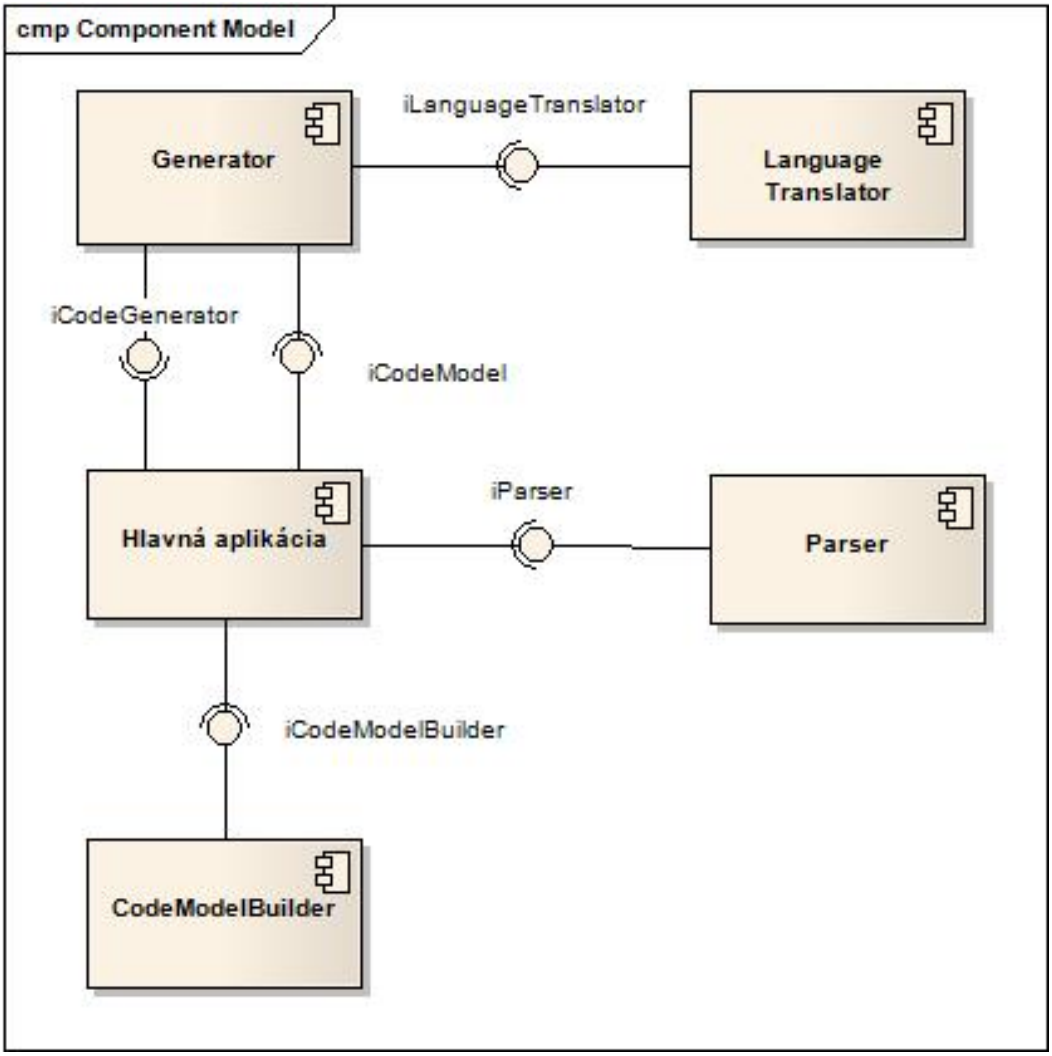
py (metód, premenných...) a hodnoty. Aby bolo ale možné jeden generator používať s viacerými rôznymi translatormi a pridávať nové translatory musí to byť práve translator, kto povie generatoru, ako sa dany typ v danom jazyku označuje. Za týmto účelom sú súčasťou rozhrania translatora metódy `translateType`, `translateValue` a `translateVisibility`, ktoré na vstupe dostanú objekt implementujúci rozhranie `IType`, rozhranie `IEvaluable` alebo hodnotu z výčtu `Visibility` a ich účelom je vrátiť string reprezentujúci daný typ, hodnotu, alebo viditeľnosť v danom jazyku. Veľmi bežné sú potom volania typu ako `Trans.PrintEqCondition(Trans.TranslateValue(ec.Left), Trans.TranslateValue(ec.Right));`, kde sa prekladajú typy či hodnoty.

Podobne sa hlavná aplikácia správa aj u načítavania parserov vstupných súborov - v podadresári `parsers` hľadá v dll súboroch implementácie rozhrania IParser a naplní zoznam vstupných formátov hodnotami, ktoré získa volaním metódy `GetHumanReadableName` na daných implementáciách. Okrem týchto hodnôt tam doplní jednu špeciálnu hodnotu a to hodnotu **autodetect**. Použitie tejto voľby je vhodné pre užívateľa, ktorý nepozná vstupný formát súboru. Hlavná aplikácia v tomto prípade postupne prejde všetky dostupné parsery a skúsi pomocou volania ich metódy `IdentifyFile` nájsť parser, ktorý daný súbor rozpoznáva. Ak sa takýto parser nájde, tak je použitý na parsovanie daného súboru. Táto metóda samozrejme nie je stopercentne úspešná. Je totiž závislá na kvalitnej odozve od parserov. Za predpokladu, že všetky dostupné parsery budú skutočne potvrdzovať len súbory, ktoré skutočne následne budú schopné parsovať, tak bude autodetect podávať správne výsledky. Ak však niektorý z parserov potvrdí, že vie daný súbor parsovať a následne ho pri samotnom parsovaní parsovať nedokáže, môže sa stať, že tým pripraví o možnosť parsovania parser, ktorý by daný formát spracovať dokázal. Manuálna voľba konkrétneho vstupného formátu je teda vždy istejšia, ale v prípade neistoty môže niekedy pomôcť funkcia `autodetect`.

Architektúru celej aplikácie a prepojenie jej jednotlivých častí zobrazuje obrázok 6.1.

6.6 Ukážková implementácia

V rámci ukážkovej implementácie bol vytvorený modul `EAPXMIParser` ako ukážková implementácia parsera a modul `CSharp` ako ukážková implementácia generujúceho modulu. Ako je jasné z názvu, modul generuje kód do jazyka C#. Ide o najjednoduchšiu variantu, teda modul, ktorý definuje len vlastnú



Obrázek 6.1: Architektúra aplikácie

implementáciu rozhrania `ILanguageTranslator`. Ostatné dve rozhrania balíčka `generator` sú zastúpené štandardnými implementáciami.

6.7 Vygenerovaný kód

V prípade použitia štandardného generátora je výstupný kód generovaný postupom popísaným v 5.3. Z pohľadu užívateľa kódu je výstupný program rozdelený na samostatné súbory pričom každý súbor obsahuje práve jednu triedu popisujúcu práve jeden automat. Táto trieda poskytuje smerom von 2 výčtový typ `events`, ktorý obsahuje udalosti, ktoré je možné automatu zasielať prostredníctvom volania metódy `processEvent`. Táto metóda vždy vráti booleanovú hodnotu informujúcu o tom, či sa daná udalosť podarila obslúžiť, alebo nie. Výčtový typ `events` je doplnený o jednu "umelú" hodnotu `__CGEN__NO__MSG__`, ktorú je možné automatu zaslať v prípade, že užívateľ chce vykonať krok automatu, ale nechce mu poslať žiadnu udalosť. To v praxi znamená, že automat sa pokúsi použiť nejaký z prechodov, ktoré nie sú spojené s nejakou udalosťou. Tieto prechody sa rovnako môžu použiť aj v prípade, ak zaslaná reálna udalosť nevedie k žiadnemu prechodu. Ďalšími verejnými metódami vygenerovanej triedy sú metódy `isIn<state>` kde `<state>` názov stavu automatu. Návratovým typom týchto metód je `boolean` a môžu byť z kódu volané pre zistenie či sa automat v danom stave aktuálne nachádza. Poslednou verejnou metódou je metóda `initialize`, ktorá musí byť zavolaná raz pred začiatkom práce s automatom (pred prvým volaním metódy `processEvent`). Jej úlohou je zinicializovať jednotlivé listy, ktoré budú udržovať kontext automatu a jeho podautomatov. V prípade, že užívateľ kódu túto metódu nezavolá manuálne zo svojho programu, tak je zavolaná automaticky pri prvom volaní metódy `processEvent`.

7. Záver

Po popísaní všetkých procesov implementácie generovania kódu zo stavového modelu UML, je čas na vyhodnotenie výsledku. Vyhodnotenie bude vykonané prostredníctvom porovnania výsledku s požiadavkami stanovenými na začiatku práce.

1. Aby generovanie nebolo úzko naviazané na jeden konkrétny výstupný programovací jazyk navrhnúť generátor tak, aby podporoval moduly umožňujúce generovanie kódu v rôznych cieľových jazykoch, aby si užívateľ mohol vybrať výstupný jazyk, prípadne si doprogramovať modul pre ďalší výstupný jazyk.

   Aplikácia pri generovaní pracuje s rozhraniami definovanými v balíčku `generator`. Pri načítaní automaticky vyhľadá ich implementácie v dostupných dll súboroch a vytvorí z nich ponuku pre užívateľa. Keď chce užívateľ vytvoriť vlastný modul, môže implementovať jedno, alebo viac z týchto rozhraní a tým definovať pravidlá generovania do iného jazyka. Cieľ bol teda naplnený.

2. Umožniť rôzne formáty vstupných súborov s tým, že formát bude automaticky detekovaný a že užívateľ si bude môcť doprogramovať vlastný parser pre ľubovoľný formát súboru.

   Cieľ bol naplnený. Aplikácia pri parsovaní vstupného súboru pracuje s rozhraním definovaným v balíčku `parser`. Pri načítaní automaticky vyhľadá jeho implementácie v dostupných dll súboroch a vytvorí z nich ponuku pre užívateľa. Užívateľ môže zvoliť daný formát manuálne, alebo použiť autodetekciu. Tá využíva informácie od parsera a je teda dôležité, aby parser vedel správne identifikovať, či je schopný so súborom pracovať. Implementovať vlastný parser je jednoduché, stačí implementovať rozhranie `IParser`.

3. Ponechať ťažisko práce a logiky na samotnom generátore tak, aby užívateľ, ktorý si chce doprogramovať generovanie pre ďalší jazyk, nemusel programovať logiku generovania kódu zo stavového automatu, ale len dodal modul ktorý bude vedieť vyjadriť základné javy kódu v danom jazyku.

   Splnenie tohto cieľa plynie z návrhu architektúry aplikácie. V štandardnej konfigurácii potrebuje programátor naprogramovať iba implementá-

ciu rozhrania `ILanguageTranslator`, ktorá obsahuje len metódy slúžiace na popis jednotlivých javov v programovacom jazyku. Tento prístup ale obmedzuje množinu možných výstupných jazykov a tak bol program navrhnutý tak, aby v prípade potreby bolo možno preimplementovať aj hlbšie súčasti systému. Aplikácia s týmto faktom pracuje automaticky a rozhoduje, či sa použijú štandardné implementácie rozhraní, alebo implementácie dodané v module. Týmto je naplnená požiadavka jednoduchosti, ale zároveň je otvorená cesta pre široké použitie.

4. Návrh overiť na prototypových moduloch, ktoré budú vedieť spracovávať jeden vstupný formát a následne generovať kostru kódu v jednom z bežne používaných jazykov. Keďže projekt je implementovaný v jazyku C#, ako ukážkový výstupný formát bol zvolený práve tento jazyk. Ako vstupný formát bol zvolený jazyk XMI (vo verzii 2.1), ktorý je štandardom pre prenos modelov v jazyku UML medzi rôznymi programami. Pre jeho rozšírenosť je najpravdepodobnejšie jeho použitie v praxi. Ostatné formáty sú voči nemu len doplnkovými a očakáva sa, že v typickom scenári použitia nebude potrebné implementovať vlastný modul pre vstupný jazyk. Z tohto dôvodu je jazyku XMI venovaná aj samostatná kapitola.

   Požadované moduly boli implementované a tak je táto požiadavka splnená. Kompletné zdrojové kódy aplikácie aj vytvorených modulov sa nachádzajú na priloženom CD. Na CD je umiestnená taktiež binárna verzia programu spolu so správne umiestnenými modulmi. Pre lepšiu orientáciu je adresárová štruktúra CD popísaná v B.1. Informácie o použití aplikácie sa nachádzajú v B.2.

Z môjho pohľadu boli požiadavky splnené a implementácia je funkčná. Aplikácia generuje použiteľný kód, ktorý je hneď pripravený na zahrnutie do programu a používanie. Z pohľadu programátora je pomerne jednoduché ju rozširovať o ďalšie vstupné formáty a výstupné jazyky.

Literatura

[1] Arlow, J., Neustat, I. *UML 2 a unifikovaný proces vývoje aplikací*. 1. vydání. Brno: Computer Press, 2007. 452 s. ISBN 978-80-251-1503-9.

[2] Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language - User Guide*. 11. vydání. Boston: Addison-Wesley, 2003. 303 s. ISBN 0-201-57168-4.

[3] *Object Management Group - UML* [online]. 2011-07-25 [cit. 2011-11-29]. Dostupné na World Wide Web: <http://www.uml.org>.

A. Ukážka funkčnosti a použitia

Spolu s plnou implementáciou generovania kódu zo stavového modelu UML bola pripravená tiež ukážka funkčnosti a použitia. Použiteľnosť výsledného generovaného kódu. Bola napísaná v jazyku C# s použitím testovacieho frameworku Visual Studio Team Test.
Ukážka je tvorená triedou CompleteStateMachineTest a nachádza sa v adresári
/CodeGen/test/samples.

A.1 Náplň ukážky

Test pracuje s kódom vygenerovaným z ukážkového vstupného automatu `ISPDialer` a teda s generovanou triedou `ISPDialer`. V setUp fáze unit testu je vytvorená inštancia `ISPDialer`. Test následne posiela automatu udalosti a opakovane sa ho pýta či sa nachádza vo finálnom stave. Keď ho dovedie do finálneho stavu, automat aj test skončia a test skončí úspechom. Ukážka použitia rozhrania a funkčnosti sa spúšťa prostredníctvom Visual Studio Team Test frameworku. K vytvoreniu a spusteniu inej aplikácie využívajúcej kód generovaný pomocou aplikácie stačí pridať triedu obsahujúcu vygenerovaný kód do svojho projektu, vytvoriť jej inštanciu a používať jej verejné metódy volaním zo svojho kódu. V prípade, že boli v zdrojovom automate použité nejaké akcie alebo podmienky, ktoré nekorenšpondujú so syntaxou daného jazyka, je potrebné výstupný kód upraviť do tej miery, aby ho bolo možné skompilovať.

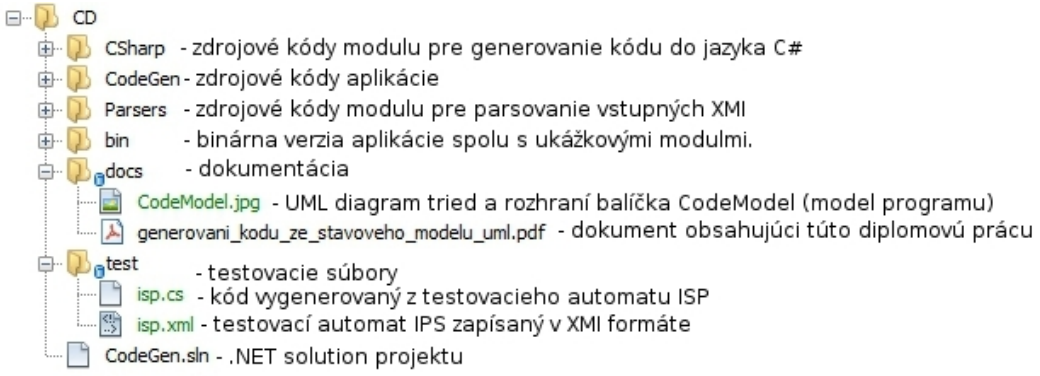
B. CD-ROM

B.1 Obsah CD-ROM

CD-ROM obsahuje túto diplomovú prácu vo formáte PDF a kompletný zdrojový kód projektu s implementovanou funkcionalitou popísanou v tejto práci. Popis štruktúry CD zobrazuje obrázok B.1.

B.2 Ako spustiť aplikáciu

Aplikácia sa spúšťa súborom `CodeGen.exe` z adresára `bin`. Pre zadanie vstupného automatu je potrebné vybrať súbor s automatom zo súborového systému pomocou dialógu na výber súborov, ktorý sa zobrazí po kliknutí na tlačítko `Browse`. Druhou možnosťou je vyplniť cestu k súboru ručne do textového poľa označeného `XMI Filename`. V rozklikávacom zozname `Input Type` je možné vybrať typ vstupného súboru. Implicitne je zvolená hodnota `autodetect`, čo znamená, že program sa pokúsi sám vybrať najvhodnejší z dostupných parserov a použiť ho. V rozklikávacom zozname `Output Type` je možné vybrať výstupný jazyk do ktorého má byť kód generovaný. Samotné generovanie sa spustí kliknutím na tlačítko `Generate Code`. Ak bolo generovanie úspešné v centrálnej časti aplikácie sa otvoria záložky s generovaným kódom - pre každú triedu automatu jedna. Uložiť do súboru kód z aktuálne otvorenej záložky je možné kliknutím na tlačítko `Save`. Užívateľovi sa zobrazí dialóg na uloženie súboru, kde môže zvoliť umiestnenie a názov nového súboru. Ak chce užívateľ uložiť všetky vygenerované súbory, môže tak vykonať kliknutím na tlačítko `Save All`, po ktorom mu je prostredníctvom zobrazeného dialógu zvoliť prie-

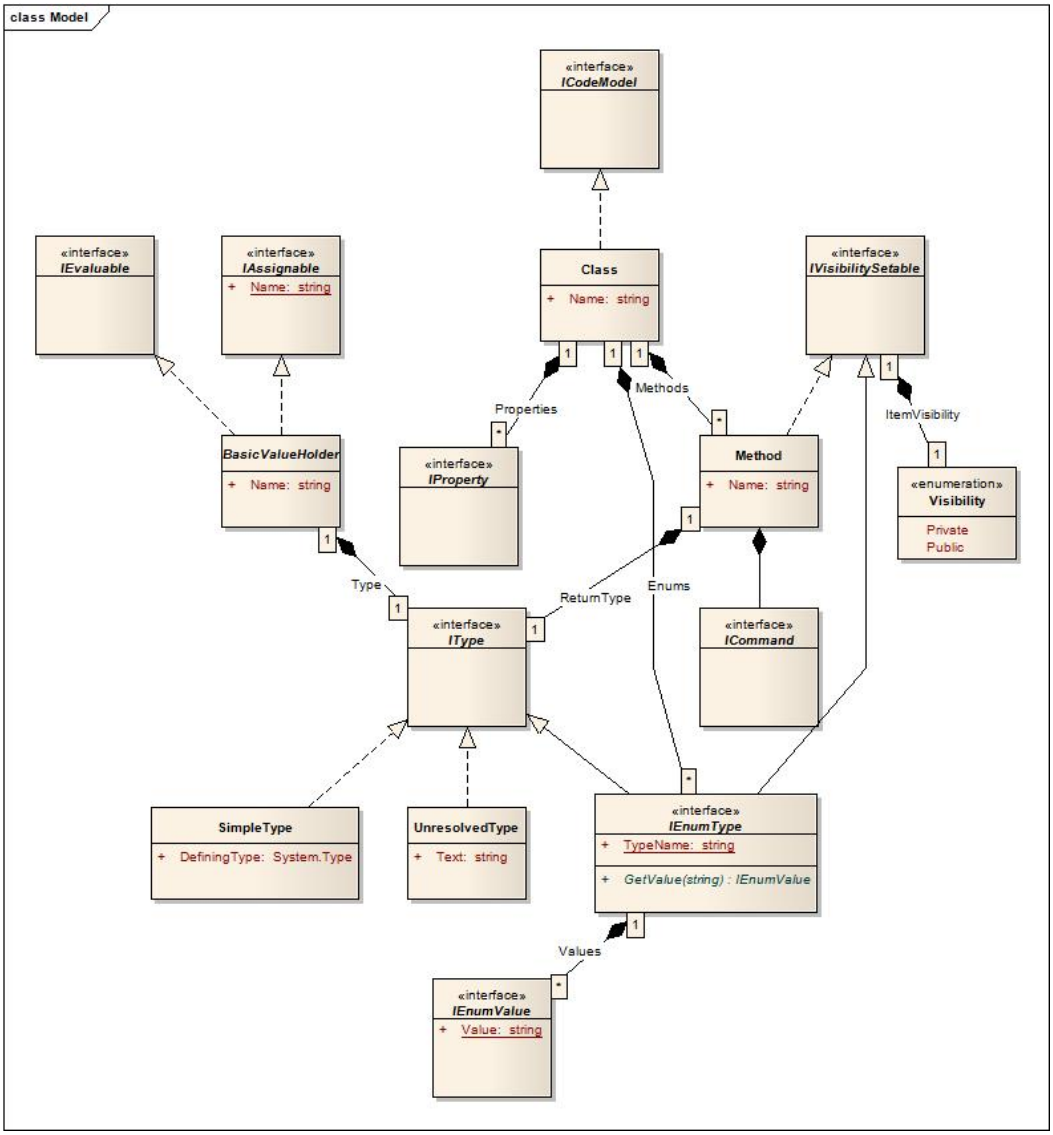


Obrázek B.1: Obsah priloženého CD

činok, do ktorého sa majú súbory uložiť. V tomto prípade sú súbory pomenovávané automaticky - totožne s názvom príslušnej triedy a je k nim pripojená prípona definovaná v príslušnej implementácii `ILanguageTranslator`.

C. Objektový model programu - balíček CodeModel

Balíček CodeModel obsahuje dovedna 40 tried a rozhraní, ktoré reprezentujú jednotlivé javy programovacieho jazyka. Tieto triedy a rozhrania sú silno prepájané dedičnosťou, implementáciou, kompozíciou či agregáciou. Nie je teda možné zobraziť celý komplexný UML diagram popisujúci celý obsah tohoto balíčka na stránkach tejto práce. Obrázok C.1 teda zobrazuje len zjednodušený fragment celého UML diagramu, z ktorého boli odstránené všetky triedy implementujúce rozhranie `ICommand`, všetky triedy dediace od triedy `BasicValue` a všetky triedy používané na reprezentáciu listových typov a prácu s nimi. Kompletný UML diagram všetkých tried a rozhraní balíčka CodeModel je uložený na priloženom CD v adresári `docs` (súbor `CodeModel.jpg`).



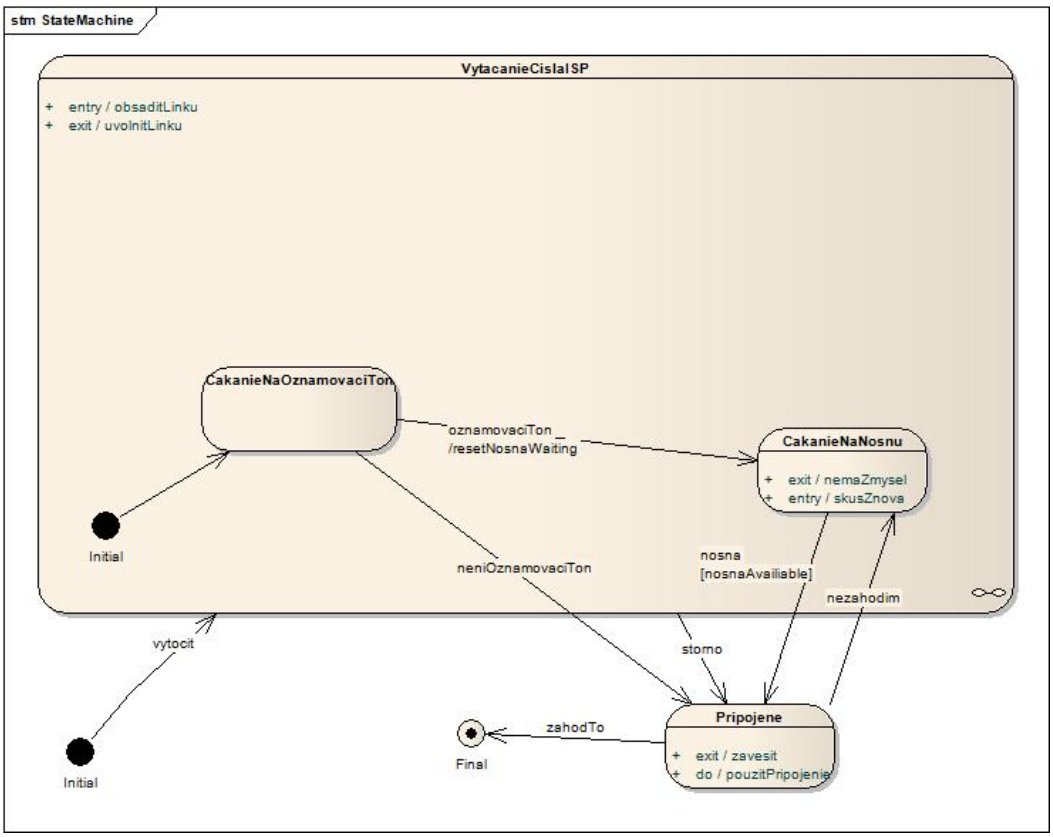
Obrázek C.1: Objektový model programu

D. Praktická ukážka

Na obrázku D.1 je zobrazený stavový automat, ktorý obsahuje iniciálny stav, jeden zložený stav a jeden jednoduchý stav. Jednoduchý stav, zložený stav aj jeden z jeho podstavov majú definované vstupné a výstupné akcie. Úmyselne sú tu vedené rozličné typy prechodov - prechod vedúci do zloženého stavu, prechody vedené vo vnútri zloženého stavu, prechod vedúci zo zloženého stavu a prechody vedúce z podstavov zloženého stavu do stavu mimo neho a opačným smerom. Týmto sú na dvoch úrovniach zastúpené prakticky všetky možnosti vedenia prechodov. Taktiež sa tu nachádzajú prechody bez udalosti aj prechody s udalosťou, niektoré prechody sú obmedzené kontrolnou podmienkou a s niektorými je spojená nejaká akcia. Cieľom bolo na malom priestore namodelovať čo najviac zaujímavých javov.

Na obrázku D.2 je zobrazený daný automat zapísaný v XMI. Z priestorových dôvodov sú z obrázka opäť vystrihnuté len tie kľúčové časti, ktoré skutočne popisujú automat. Celý kód je možné nájsť na priloženom CD v adresári `test` (súbor `isp.xml`).

Na obrázku D.3 je zobrazený kód metódy `processEvent`, ktorá obsluhuje chod daného automatu. Celý vygenerovaný kód obsahujúci celú triedu a všetky vygenerované metódy sa opäť na priloženom CD v adresári `test` (súbor `isp.cs`).



Obrázek D.1: Diagram testovacieho automatu

```
<subvertex xmi:type="uml:State" xmi:id="EAID_27FBDFB7_C00B_4e5b_B2C4_4AA7E453ABB1" name="Pripojene" visibility="public" isSubmachineState="false">
  <incoming xmi:idref="EAID_CD02F94F_C070_464b_A44C_B5EFDD8411B0"/>
  <incoming xmi:idref="EAID_BAE3255D_AAB6_4e19_9FB2_F796703DE192"/>
  <incoming xmi:idref="EAID_07862869_28B0_4f90_A29C_B2911F633FCC"/>
  <outgoing xmi:idref="EAID_AF0ACA97_EAAB_4742_BD8A_1EE7627BD0ED"/>
  <outgoing xmi:idref="EAID_0ACAE070_32BB_4522_9084_2BD6BF8A5A40"/>
  <exit xmi:type="uml:Behavior" xmi:id="EAID_BH000000_CDA1_422b_AE64_5A257286927E" visibility="public">
    <ownedOperation xmi:id="EAID_992FC5CE_CDA1_422b_AE64_5A257286927E" name="zavesit" visibility="public" concurrency="sequential">
      <ownedParameter xmi:id="EAID_RT000000_CDA1_422b_AE64_5A257286927E" name="return" direction="return" type="EAJava_exit"/>
    </ownedOperation>
  </exit>
  <doActivity xmi:type="uml:Behavior" xmi:id="EAID_BH000000_4AAB_460a_813F_425B2758B727" visibility="public">
    <ownedOperation xmi:id="EAID_90B575E6_4AAB_460a_813F_425B2758B727" name="pouzitPripojenie" visibility="public" concurrency="sequential">
      <ownedParameter xmi:id="EAID_RT000000_4AAB_460a_813F_425B2758B727" name="return" direction="return" type="EAJava_do"/>
    </ownedOperation>
  </doActivity>
</subvertex>
<transition xmi:type="uml:Transition" xmi:id="EAID_0ACAE070_32BB_4522_9084_2BD6BF8A5A40" visibility="public" kind="local"
            source="EAID_27FBDFB7_C00B_4e5b_B2C4_4AA7E453ABB1" target="EAID_4CE8E9D3_5EEE_425e_8CD1_C6B3CD3BDC49">
  <trigger xmi:idref="EAID_3FD1433D_6B36_4405_84A6_A27AF397A67A"/>
</transition>
<transition xmi:type="uml:Transition" xmi:id="EAID_AF0ACA97_EAAB_4742_BD8A_1EE7627BD0ED" visibility="public" kind="local"
            source="EAID_27FBDFB7_C00B_4e5b_B2C4_4AA7E453ABB1" target="EAID_FABC0896_3BBC_4506_9DA8_B4C6D9BE33B3">
  <trigger xmi:idref="EAID_9352DFD4_2201_4ff9_B00B_B57ABE878CBD"/>
</transition>
<subvertex xmi:type="uml:State" xmi:id="EAID_5563B2E1_FDB4_4054_8190_36D49280172C" name="VytacanieCislaISP" visibility="public" isSubmachineState="true">
  <incoming xmi:idref="EAID_BB8DBAE7_F769_4850_8FE9_DBFE6AE178A1"/>
  <outgoing xmi:idref="EAID_07862869_28B0_4f90_A29C_B2911F633FCC"/>
  <entry xmi:type="uml:Behavior" xmi:id="EAID_BH000000_A64E_48c0_8EC5_211A285F8D0D" visibility="public">
    <ownedOperation xmi:id="EAID_00D00185_A64E_48c0_8EC5_211A285F8D0D" name="obsaditLinku" visibility="public" concurrency="sequential">
      <ownedParameter xmi:id="EAID_RT000000_A64E_48c0_8EC5_211A285F8D0D" name="return" direction="return" type="EAJava_entry"/>
    </ownedOperation>
  </entry>
  <exit xmi:type="uml:Behavior" xmi:id="EAID_BH000000_7D45_4a43_87D1_1806B933FC33" visibility="public">
    <ownedOperation xmi:id="EAID_3E72C6F4_7D45_4a43_87D1_1806B933FC33" name="uvolnitLinku" visibility="public" concurrency="sequential">
      <ownedParameter xmi:id="EAID_RT000000_7D45_4a43_87D1_1806B933FC33" name="return" direction="return" type="EAJava_exit"/>
    </ownedOperation>
  </exit>
  <region xmi:type="uml:Region" xmi:id="EAID_SR000002_54BB_45bf_A1F5_D36D5F1B38ED" name="EA_Region2" visibility="public">
    <subvertex xmi:type="uml:State" xmi:id="EAID_FABC0896_3BBC_4506_9DA8_B4C6D9BE33B3" name="CakanieNaNosnu" visibility="public" isSubmachineState="false">
      <incoming xmi:idref="EAID_DE7D6341_ACFC_4faf_9F96_5C9A2EE5F692"/>
      <incoming xmi:idref="EAID_AF0ACA97_EAAB_4742_BD8A_1EE7627BD0ED"/>
      <outgoing xmi:idref="EAID_CD02F94F_C070_464b_A44C_B5EFDD8411B0"/>
      <exit xmi:type="uml:Behavior" xmi:id="EAID_BH000000_AC42_4632_AA53_6DFF3A23F6C7" visibility="public">
        <ownedOperation xmi:id="EAID_B42FD180_AC42_4632_AA53_6DFF3A23F6C7" name="nemaZmysel" visibility="public" concurrency="sequential">
          <ownedParameter xmi:id="EAID_RT000000_AC42_4632_AA53_6DFF3A23F6C7" name="return" direction="return" type="EAJava_exit"/>
        </ownedOperation>
      </exit>
      <entry xmi:type="uml:Behavior" xmi:id="EAID_BH000000_CE43_4396_BE24_28860BD3E095" visibility="public">
        <ownedOperation xmi:id="EAID_AD9EF68C_CE43_4396_BE24_28860BD3E095" name="skusZnova" visibility="public" concurrency="sequential">
          <ownedParameter xmi:id="EAID_RT000000_CE43_4396_BE24_28860BD3E095" name="return" direction="return" type="EAJava_entry"/>
        </ownedOperation>
      </entry>
    </subvertex>
    <transition xmi:type="uml:Transition" xmi:id="EAID_CD02F94F_C070_464b_A44C_B5EFDD8411B0" name=" [nosna]" visibility="public" kind="local"
                source="EAID_FABC0896_3BBC_4506_9DA8_B4C6D9BE33B3" target="EAID_27FBDFB7_C00B_4e5b_B2C4_4AA7E453ABB1">
      <trigger xmi:idref="EAID_817B61CA_5359_4456_9F80_4FD62BB39D1E"/>
      <guard xmi:type="uml:Constraint" xmi:id="EAID_C0000000_C070_464b_A44C_B5EFDD8411B0">
        <specification xmi:type="uml:OpaqueExpression" xmi:id="EAID_OE000000_C070_464b_A44C_B5EFDD8411B0" body="nosnaAvailiable"/>
      </guard>
    </transition>
    <subvertex xmi:type="uml:State" xmi:id="EAID_DD4A4728_CBAA_42ba_877D_3D76902D2630" name="CakanieNaOznamovaciTon" visibility="public" isSubmachineState="false">
      <incoming xmi:idref="EAID_7D26F79B_29FE_413d_B947_B44B22B44398"/>
      <outgoing xmi:idref="EAID_DE7D6341_ACFC_4faf_9F96_5C9A2EE5F692"/>
      <outgoing xmi:idref="EAID_BAE3255D_AAB6_4e19_9FB2_F796703DE192"/>
    </subvertex>
    <transition xmi:type="uml:Transition" xmi:id="EAID_BAE3255D_AAB6_4e19_9FB2_F796703DE192" visibility="public" kind="local"
                source="EAID_DD4A4728_CBAA_42ba_877D_3D76902D2630" target="EAID_27FBDFB7_C00B_4e5b_B2C4_4AA7E453ABB1">
      <trigger xmi:idref="EAID_1740BF8A_BF20_40e8_9878_8404EC08CF47"/>
    </transition>
    <transition xmi:type="uml:Transition" xmi:id="EAID_DE7D6341_ACFC_4faf_9F96_5C9A2EE5F692" visibility="public" kind="local"
                source="EAID_DD4A4728_CBAA_42ba_877D_3D76902D2630" target="EAID_FABC0896_3BBC_4506_9DA8_B4C6D9BE33B3">
      <trigger xmi:idref="EAID_36F01AE1_1B9D_4b9d_AE88_0F65ECFF994D"/>
      <effect xmi:type="uml:OpaqueBehavior" xmi:id="EAID_OB000000_ACFC_4faf_9F96_5C9A2EE5F692" body="resetNosnaWaiting"/>
    </transition>
    <subvertex xmi:type="uml:Pseudostate" xmi:id="EAID_94B6F9A2_2CBC_481d_94E3_79987AE231C7" name="Initial" visibility="public" kind="initial">
      <outgoing xmi:idref="EAID_7D26F79B_29FE_413d_B947_B44B22B44398"/>
    </subvertex>
    <transition xmi:type="uml:Transition" xmi:id="EAID_7D26F79B_29FE_413d_B947_B44B22B44398" visibility="public" kind="local"
                source="EAID_94B6F9A2_2CBC_481d_94E3_79987AE231C7" target="EAID_DD4A4728_CBAA_42ba_877D_3D76902D2630"/>
  </region>
</subvertex>
<transition xmi:type="uml:Transition" xmi:id="EAID_07862869_28B0_4f90_A29C_B2911F633FCC" visibility="public" kind="local"
            source="EAID_5563B2E1_FDB4_4054_8190_36D49280172C" target="EAID_27FBDFB7_C00B_4e5b_B2C4_4AA7E453ABB1">
  <trigger xmi:idref="EAID_A4B7E73D_7237_4692_ADA9_95103598D444"/>
</transition>
<subvertex xmi:type="uml:FinalState" xmi:id="EAID_4CE8E9D3_5EEE_425e_8CD1_C6B3CD3BDC49" name="Final" visibility="public">
  <incoming xmi:idref="EAID_0ACAE070_32BB_4522_9084_2BD6BF8A5A40"/>
</subvertex>
<subvertex xmi:type="uml:Pseudostate" xmi:id="EAID_D0FB756A_B6AA_4043_BA81_F351B96A09D4" name="Initial" visibility="public" kind="initial">
  <outgoing xmi:idref="EAID_BB8DBAE7_F769_4850_8FE9_DBFE6AE178A1"/>
</subvertex>
<transition xmi:type="uml:Transition" xmi:id="EAID_BB8DBAE7_F769_4850_8FE9_DBFE6AE178A1" visibility="public" kind="local"
            source="EAID_D0FB756A_B6AA_4043_BA81_F351B96A09D4" target="EAID_5563B2E1_FDB4_4054_8190_36D49280172C">
  <trigger xmi:idref="EAID_C692E11C_665C_4133_A137_48FB1E7EC173"/>
</transition>
</region>
<nestedClassifier xmi:type="uml:Trigger" xmi:id="EAID_1740BF8A_BF20_40e8_9878_8404EC08CF47" name="neniOznamovaciTon" visibility="public">
  <event xmi:type="uml:SignalEvent" xmi:id="EAID_EX000000_BF20_40e8_9878_8404EC08CF47">
    <signal xmi:idref="EAID_05A70E67_8CAD_4777_BE49_25A0EE49DBFD"/>
  </event>
</nestedClassifier>
<nestedClassifier xmi:type="uml:Trigger" xmi:id="EAID_9352DFD4_2201_4ff9_B00B_B57ABE878CBD" name="nezahodim" visibility="public">
  <event xmi:type="uml:SignalEvent" xmi:id="EAID_EX000000_2201_4ff9_B00B_B57ABE878CBD">
    <signal xmi:idref="EAID_34D83A76_E55E_4110_A32F_40B7DC9A7C40"/>
  </event>
</nestedClassifier>
<nestedClassifier xmi:type="uml:Trigger" xmi:id="EAID_817B61CA_5359_4456_9F80_4FD62BB39D1E" name="nosna" visibility="public">
  <event xmi:type="uml:SignalEvent" xmi:id="EAID_EX000000_5359_4456_9F80_4FD62BB39D1E">
    <signal xmi:idref="EAID_64AF4705_8E6F_40df_BCE7_4C5FB196A4CB"/>
  </event>
</nestedClassifier>
<nestedClassifier xmi:type="uml:Trigger" xmi:id="EAID_36F01AE1_1B9D_4b9d_AE88_0F65ECFF994D" name="oznamovaciTon" visibility="public">
    <event xmi:type="uml:SignalEvent" xmi:id="EAID_EX000000_1B9D_4b9d_AE8B_0F65ECFF994D">
      <signal xmi:idref="EAID_BEFCBE89_B17D_496c_AF4D_11477C22F84B"/>
    </event>
  </nestedClassifier>
```

Obrázek D.2: Testovací automat zapísaný v XMI

```
public bool processEvent(events e)
{
    bool eventConsumed;
    eventConsumed = false;
    if (initialized == false)
    {
        initialize();
    }
    if (currentState.Contains(states.Initial))
    {
        if (e == events.vytocit)
        {
            eventConsumed = true;
            currentState.Remove(states.Initial);
            currentState.Add(states.VytacanieCislaISP);
            VytacanieCislaISPEntryAction();
            return true;
        }
    }
    if (currentState.Contains(states.Final))
    {
    }
    if (currentState.Contains(states.Pripojene))
    {
        if (e == events.zahodTo)
        {
            eventConsumed = true;
            PripojeneExitAction();
            currentState.Remove(states.Pripojene);
            currentState.Add(states.Final);
            return true;
        }
        if (e == events.nezahodim)
        {
            eventConsumed = true;
            PripojeneExitAction();
            currentState.Remove(states.Pripojene);
            currentState.Add(states.VytacanieCislaISP);
            currentStateVytacanieCislaISP.Remove(states.Initial_EAID_94B6F9A2_2CBC_481d_94E3_79987AE231C7);
            currentStateVytacanieCislaISP.Add(states.CakanieNaNosnu);
            VytacanieCislaISPEntryAction();
            CakanieNaNosnuEntryAction();
            return true;
        }
    }
    if (currentState.Contains(states.VytacanieCislaISP))
    {
        if (currentStateVytacanieCislaISP.Contains(states.Initial_EAID_94B6F9A2_2CBC_481d_94E3_79987AE231C7))
        {
            if (eventConsumed == false)
            {
                eventConsumed = true;
                currentStateVytacanieCislaISP.Remove(states.Initial_EAID_94B6F9A2_2CBC_481d_94E3_79987AE231C7);
                currentStateVytacanieCislaISP.Add(states.CakanieNaOznamovaciTon);
                return true;
            }
        }
        if (currentStateVytacanieCislaISP.Contains(states.CakanieNaNosnu))
        {
            if (e == events.nosna)
            {
                if (nosnaAvailiable)
                {
                    eventConsumed = true;
                    CakanieNaNosnuExitAction();
                    VytacanieCislaISPExitAction();
                    currentStateVytacanieCislaISP.Remove(states.CakanieNaNosnu);
                    currentStateVytacanieCislaISP.Add(states.Initial_EAID_94B6F9A2_2CBC_481d_94E3_79987AE231C7);
                    currentState.Remove(states.VytacanieCislaISP);
                    currentState.Add(states.Pripojene);
                    return true;
                }
            }
        }
        if (currentStateVytacanieCislaISP.Contains(states.CakanieNaOznamovaciTon))
        {
            if (e == events.neniOznamovaciTon)
            {
                eventConsumed = true;
                VytacanieCislaISPExitAction();
                currentStateVytacanieCislaISP.Remove(states.CakanieNaOznamovaciTon);
                currentStateVytacanieCislaISP.Add(states.Initial_EAID_94B6F9A2_2CBC_481d_94E3_79987AE231C7);
                currentState.Remove(states.VytacanieCislaISP);
                currentState.Add(states.Pripojene);
                return true;
            }
            if (e == events.oznamovaciTon)
            {
                eventConsumed = true;
                CakanieNaOznamovaciTonCakanieNaNosnuAction();
                currentStateVytacanieCislaISP.Remove(states.CakanieNaOznamovaciTon);
                currentStateVytacanieCislaISP.Add(states.CakanieNaNosnu);
                CakanieNaNosnuEntryAction();
                return true;
            }
        }
        if (eventConsumed == false)
        {
            if (e == events.storno)
            {
                eventConsumed = true;
                if (currentStateVytacanieCislaISP.Contains(states.CakanieNaNosnu))
                {
                    CakanieNaNosnuExitAction();
                }
                VytacanieCislaISPExitAction();
                currentStateVytacanieCislaISP.Clear();
                currentStateVytacanieCislaISP.Add(states.Initial_EAID_94B6F9A2_2CBC_481d_94E3_79987AE231C7);
                currentState.Remove(states.VytacanieCislaISP);
                currentState.Add(states.Pripojene);
                return true;
            }
        }
    }
    return false;
}
```

Obrázek D.3: Kód vygenerovaný z testovacieho automatu